Simulador DLX con repertorio multimedia

Jaime Márquez Pascual
Daniel Ochayta Martín
Blanca Puentes Ramiro

Director
Carlos García Sánchez
Sistemas Informáticos
Curso 2010-2011
Facultad de Informática
Universidad Complutense de Madrid

Índice

Declaración de conformidad

Los alumnos:
Jaime Márquez Pascual, Daniel Ochayta Martín, Blanca Puentes Ramiro abajo firmantes autorizan a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la propia memoria, como el código, la documentación y/o el prototipo desarrollado.

Madrid, 7 de Julio de 2011

Jaime Márquez Pascual

Daniel Ochayta Martín

Blanca Puentes Ramiro

Agradecimientos

Blanca:

«Gli amici di sempre, gli abbracci più lunghi, la musica, aprire i regali. I viaggi lontani che fanno sognare. I film che ti restano impressi nel cuore. Gli sguardi e quell'attimo prima di un bacio, le stelle cadenti, il profumo del vento. La vita rimane la cosa più bella che ho… E da qui non c'è niente di più naturale, che fermarsi un momento a pensare che le piccole cose son quelle più vere e restano dentro di te»

Nek, “E da qui”, Greatest Hits 1992-2010 (2010)

Gracias a los buenos amigos, a los que han estado en los peores momentos de este año, sobre todo en los días agobiados, a causa de este proyecto, y por supuesto por todos los buenos momentos que hemos pasado juntos. En general, gracias a los amigos, tanto a los de siempre, como a los que he ido conociendo a lo largo de la carrera.

Gracias a mi familia por el apoyo para seguir adelante durante toda la carrera, en especial a mis padres, ya que sin ellos no está claro que hubiera sido posible llegar hasta aquí.

Por último, y no por ello menos importante, también agradecer el trabajo realizado por parte de nuestro director de proyecto Carlos. Hicimos muy bien en “escogerle” como tutor para el proyecto más importante de la carrera.

Daniel:

«No conoceréis al miedo. El miedo mata la mente. El miedo es la pequeña muerte que conduce a la destrucción total. Afrontaré mi miedo. Permitirá que pase sobre mí y a través de mí. Y cuando haya pasado, giraré mi ojo interior para escrutar su camino. Allá donde haya pasado el miedo ya no habrá nada. Sólo estaré yo.»

Dune (1965), Frank Herbert

Quiero agradecérselo a mis dos compañeros que han estado conmigo a lo largo del curso y que sin ellos no habría sido posible realizar el proyecto, mi genio y mi dedicación, porque la vida sin humor carece de sentido.

Y ahora seriamente, agradecérselo a mis compañeros, Jaime y Blanca que aportaron me acompañaron en esta dura tarea.

Dar las gracias también a nuestro tutor Carlos, que su tiempo nos ha dedicado y cuyo humor en las correcciones no tiene igual.

Finalmente, gracias especialmente a la señorita Blanca Puentes Ramiro por el tiempo que pasamos juntos con y sin el proyecto al mostrarme que la vida son algo más que 1's y 0's, pero no mucho más.

Jaime:

«Cuando el cuerpo dice basta la mente dice adelante»

Anónimo

Quiero agradecerle a mi familia por ser un apoyo constante y diario durante toda la carrera. A Carlos por todo el tiempo que ha pasado buscando información y atendiendo y resolviendo nuestras dudas. A mis amigos que me han animado y ayudado durante este año tan duro y ajetreado y a mis amigos y compañeros de proyecto porque sin ellos hubiera sido dificilísimo sacar adelante este proyecto. Gracias a todos!

Resumen

El objetivo del proyecto era desarrollar una herramienta didáctica sobre el procesador DLX incluyendo las instrucciones multimedia mediante la incorporación del repertorio AltiVec al conjunto de instrucciones DLX. Para ello se ha implementado en Java un simulador DLX-AltiVec que permite visualizar el comportamiento de un código fuente de instrucciones del repertorio DLX y AltiVec, en un procesador DLX, mediante un pipeline, un gráfico que muestra la ruta de datos así como el estado del banco de registros y la memoria de datos. De esta manera se convierte el proyecto en una herramienta completa para el aprendizaje de los procesadores DLX y del comportamiento de instrucciones multimedia.

Abstract

The aim of the project is to develop an educational tool which simulates DLX processor including multimedia ISA (Instructions Set Architecture) available nowadays in most general purpose processors. Our simulator named DLX-AltiVec includes AltiVec instruction set and its specific hardware. Simulator has been implemented in Java to use in most of platforms. Its offers a console with a source code where users can edit and modify it before compiling, displays evolution of instructions pipeline, show memory and register in different format (hexadecimal, decimal, floating-point). For our point of view, DLX-AltiVec is a powerful tool for educational purposes because it allows to consolidate computer design and architecture knowledge.

Capítulo 1

Introducción y Motivación

Índice:

Hemos dividido este capítulo introductorio en tres secciones. En la primera de ellas se hace un resumen de la evolución que han sufrido los computadores de altas prestaciones durante la última década, centrándonos posteriormente en los avances y la consolidación de extensiones multimedia en los repertorios de instrucciones de los procesadores de propósito general. La seccione dos hace una revisión de los repertorios multimedia incorporados a los procesadores de propósito general. La sección tres presenta la motivación de este trabajo y finalmente se detallan los objetivos perseguidos.

1.1 Introducción. Tendencias en el Diseño de Microprocesadores

Año tras año venimos asistiendo a la mejora, tanto en prestaciones como en capacidad, de los sistemas basados en microprocesador. Esta mejora ha venido impulsada por los importantes avances en la tecnología de fabricación de los circuitos integrados, recogida en la famosa ley de Moore. La tecnología ha permitido una reducción exponencial en los tamaños mínimos de fabricación, con las consiguientes

mejoras en velocidad y en densidad de integración. Asimismo, ha sido posible la fabricación de chips de mayores dimensiones. El efecto combinado de ambas tendencias, es decir, el aumento en la densidad de transistores y en el tamaño de los chips, ha dado lugar a un crecimiento exponencial en el número total de transistores por chip que ronda anualmente el 55% [1] [2].
En la figura 1 se recoge esta evolución, particularizada para los procesadores comercializados por Intel Corporation[1].

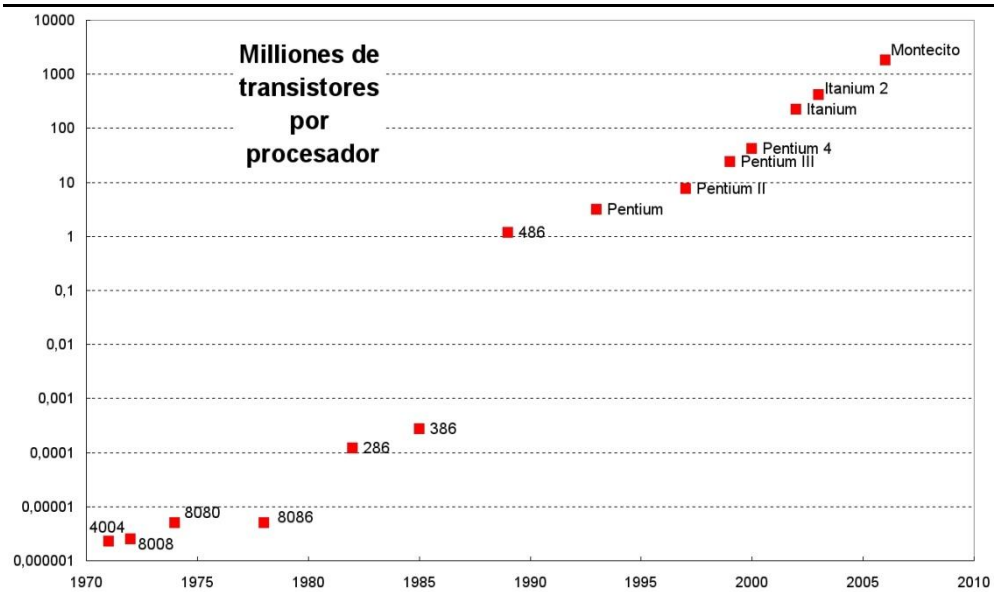


Figura 1: Evolución del número de transistores que se han integrado en las sucesivas generaciones de procesadores de Intel.

En la actualidad Intel ha anunciado la construcción de procesadores basados en tecnología de 22nm [3], y sus perceptivas son de ir disminuyendo el tamaño del mismo en los próximos años. Así se desprende de información en [3] cuya previsión en los roadmap de Intel inciden en disminuciones del tamaño del transistor hasta 10nm en 2015. Obviamente este avance tecnológico supone nuevos retos [4] [5] [2] que debe de afrontar la comunidad que se dedica a diseñar microprocesadores.
Desde el punto de vista tecnológico, destaca el consumo y la disipación de energía. Inicialmente, era un problema exclusivo del segmento de los dispositivos móviles, donde la capacidad limitada de las baterías con que operan dichos dispositivos impone restricciones muy severas. Sin embargo, actualmente ha pasado a ser un problema en todos los segmentos del mercado. En los procesadores de gama alta, ronda los 100-

1 Según los datos facilitados por *MDRonline* (editor de *Microprocessor Report*) en su "*Chart Watch: Server Processor*" de Agosto de 2006, el *Intel Itanium 2 9050* (*Montecito*) [McBh04] es el primer procesador comercial que supera la barrera del billón (americano) de transistores (1792 millones de transistores en un dado 596 mm^2).

130W, estimándose del orden de 300W para el año 2016 [6]. Además, el hecho de que los transistores sean cada vez más pequeños está provocando crecimientos insostenibles en la densidad de potencia (energía disipada por unidad de tiempo y unidad de superficie), aproximándonos en la actualidad al centenar de W/cm^2. La densidad de potencia tiene repercusiones directas en la temperatura del procesador.

Dado que el calor generado necesita disiparse para evitar posibles errores o daños en los circuitos, es necesario incluir sistemas de refrigeración y encapsulados cada vez más sofisticados, que incrementan el coste de los microprocesadores y de los sistemas en general.

No obstante, no es el único reto tecnológico al que deberemos enfrentarnos. La reducción de la escala de integración hace los chips más vulnerables a diferentes fallos transitorios y variaciones en sus parámetros característicos ("in-die process variation"). El reto que se presenta se encuentra tanto a nivel de circuito como a nivel de sistema [7] [8]. Recientemente ha adquirido especial relevancia el tratamiento de los denominados errores suaves ("soft-errors"), fallos transitorios inducidos por la radiación, causados por neutrones procedentes de rayos cósmicos y partículas "alpha" de los materiales utilizados en el empaquetado ("packaging"). Tradicionalmente, sólo representaban un problema importante en dispositivos para la industria aeroespacial. Sin embargo, en la era de la nanotecnología, este tipo de errores empiezan a ser más frecuente, y es necesario incorporar mecanismos de tolerancia a fallos para mejorar la robustez [9].

Por último, aunque no menos importante, el mayor tamaño de los chips, junto a la disminución tanto del tiempo de ciclo, como del tamaño de los transistores, dificulta la distribución de la señal de reloj [10] y provoca un aumento significativo en los retardos de las interconexiones respecto a los retardos de la lógica. Este efecto comienza a ser un problema en los sistemas multicore donde mover datos de unos componentes a otros del chip ya no puede hacerse en muchos casos en único ciclo de reloj y es necesario considerar diseños alternativos más modulares, en los que se premien comunicaciones locales [11].

1.2 Extensiones Multimedia

Índice:

1.2.1 Innovaciones en la Arquitectura del Repertorio de Instrucciones

A comienzos de la década de los 90 asistimos a un intenso debate respecto a la arquitectura del repertorio de instrucciones (ISA siglas de "Instruction Set Architecture"), los repertorios CISC frente a los RISC [1]. Sin embargo, con la aparición de los sistemas de decodificación utilizados por Intel y AMD para convertir las instrucciones CSIC del antiguo repertorio x86 en micro-operaciones tipo RISC [12], y las técnicas de traducción binaria dinámicas[2] ("Dynamic Binary Translation") [13] [14] [15], aquellas intensas discusiones quedaron relegadas a un segundo plano.
La única innovación en los repertorios de instrucciones que ha tenido éxito comercial durante esta última década ha sido las extensiones multimedia. Dichas extensiones surgen en los 90 para dar repuesta a la creciente demanda en aplicaciones multimedia (video, audio, imagen, Gráficos 3D, etc.), ya que las características ligadas a este tipo de procesamiento [16] difieren, en general, de las aplicaciones que han servido tradicionalmente de guía en el diseño de los procesadores de propósito general.

1.2.2 Características de las Aplicaciones Multimedia

Aunque algunas aplicaciones multimedia fueron incorporadas a "benchmarks" considerados como enteros (por ejemplo "eon" en los SPECint2000 o "ijpeg" en SPECint95), estas aplicaciones comparten una serie de peculiaridades que permite distinguirlas claramente, tanto de las ampliaciones en punto-flotante, como del resto de aplicaciones enteras. Entre otras características podemos destacar [17]:

2 Tuvo gran repercusión mediática el code morphing utilizado por Transmeta en el Crusoe.

- **Tipos de datos cortos**. Los sentidos del ser humano poseen un rango limitado a la hora de discretizar las variaciones en un fenómeno físico. Por ello, los datos con los que trabajan las aplicaciones multimedia no necesitan de una gran precisión[3] y se pueden utilizar tipos de datos relativamente cortos (pixels de 8 bits, señal de audio de 16 bits,...). Las rutas de datos de los procesadores de propósito general están sin embargo optimizadas para trabajar con mayor precisión. Ello supone un desperdicio de los recursos, tanto de cálculo (ALUs), como de almacenamiento (banco de registros).

- **Tiempo real.** Muchas aplicaciones multimedia requieren respuesta en tiempo real. En las aplicaciones científico-técnicas no se suele presentar este tipo de respuestas y es habitual que los diseñadores primen la productividad (el número de simulaciones por unidad de tiempo que es posible procesar).

- **Regularidad y paralelismo de datos.** Los cómputos suelen ser regulares y sencillos, aplicándose con frecuencia las mismas operaciones (a menudo se suele hablar de kernels) a un flujo de datos continuo (stream). Existe por tanto, un gran paralelismo de datos (DLP: "Data Level Parallelism") inherente, que se adapta de forma natural al paradigma SIMD (SIMD: "Single Instruction Multiple Data").

- **Escasa localidad temporal.** El volumen de datos en estas aplicaciones suele ser elevado y el reúso suele ser muy limitado. Por este motivo es fundamental disponer técnicas que permitan ocultar la latencia de los accesos a memoria.

Para tratar eficientemente estas nuevas cargas de trabajo, en la década de los 90 se extendieron las ISA de las arquitecturas de propósito general con extensiones específicas para procesamiento multimedia (extensiones multimedia), que permitían procesamiento *SIMD*.

1.2.3 Primera Generación de Extensiones Multimedia

El procesador i860 de Intel [18] fue el primer procesador de propósito general en el que se incluyen operaciones específicas (6 instrucciones) para el procesamiento de pixels (sumas) y el manejo del Z-buffer. El objetivo perseguido era la aceleración del proceso de renderización de gráficos 3D. Poco después vendría la respuesta de

3 Aunque a veces se lleven a cabo cálculos intermedios con mayor precisión.

Motorola, que en el MC88110 [19] incorpora 9 instrucciones para el manejo de gráficos, entre las que se incluyen la suma, la resta la multiplicación y el empaquetado y desempaquetado de pixels.
En estos dos primeros antecedentes, muy enfocados a la aceleración del pipeline gráfico y no al procesamiento multimedia general, la capacidad de procesamiento SIMD era muy limitada. Unos años más tarde sin embargo, Hewlett-Packard introduce las primeras extensiones multimedia en el PA-7100LC, a las que denominó MAX ("Multimedia Acceleration eXtensions"). El PA-7100LC era un procesador con arquitectura PA-RISC de 32 bits, pero las extensiones MAX operaban con dos componentes de 16 bits en paralelo. El número de operaciones soportado era muy reducido (5 instrucciones: suma (HADD), resta (HSUB) y tres sumas con distintos desplazamientos (HAVE, HSRKADD, HSLKADD)), pero a pesar de ello consiguieron importantes mejoras en el procesamiento del MPEG-1 [20].
Unos años más tarde aparecerían las extensiones las VIS ("Visual Instruction Set") de SUN para la arquitectura UltraSparc I [21], las MAX-2 de HP para arquitectura PA-RISC 2.0 [22], o las MMX ("MultiMedia eXtensions") de Intel para arquitectura Pentium. Aunque existen diferencias, la idea fundamental que subyace a todas ellas es la misma: dotar a las unidades funcionales aritmético-lógicas de la capacidad de procesamiento SIMD y poder operar simultáneamente con varios datos más cortos (sub-palabras) empaquetados en la palabra del procesador. Con ello se persigue aprovechar toda la anchura, tanto de dichas unidades, como de las rutas de datos y de los registros.
Para simplificar la integración con el sistema operativo y minimizar el coste hardware, estas primeras extensiones no utilizan un banco de registros adicional específico, sino que reúsan los registros en punto-flotante. La complejidad que es necesario añadir en las unidades funcionales para incorporar esta capacidad de procesamiento depende del tipo de operación que consideremos. Como se muestra en la figura 2, en un sumador/restador entero es relativamente sencillo: sólo es necesario romper, en los puntos apropiados, la cadena del acarreo.

Soportar paralelismo SIMD en las operaciones lógicas (and, or, xor, nand...) también es relativamente sencillo, ya que al efectuarse bit a bit, son independientes del tipo de datos utilizado y es lo mismo aplicarlas a palabras de m bits, que a cualquier tipo de datos más corto.
Las multiplicaciones enteras plantean más problemas, ya que al multiplicar dos números naturales de k bits, se necesitan (en general) 2k bits para representar el resultado, no existiendo un consenso entre las distintas extensiones. El repertorio MMX por ejemplo, soporta varios tipos distintos de multiplicación entera. En una de ellas se utiliza un multiplicador de kxk bits pero se selecciona que parte del resultado se guardará en el operando destino (la más o la menos significativa). También es posible promocionar los operandos fuente previamente a 2k bits (utilizando

instrucciones adicionales de conversión) y efectuar las multiplicaciones con esta nueva precisión, ya que en este caso no se corre el riesgo de desbordamiento. En los repertorios MAX y MAX-2 de HP sin embargo, no se incluyen extensiones de multiplicación, ya que se perseguía un diseño minimalista acorde con la filosofía RISC[4] [22].

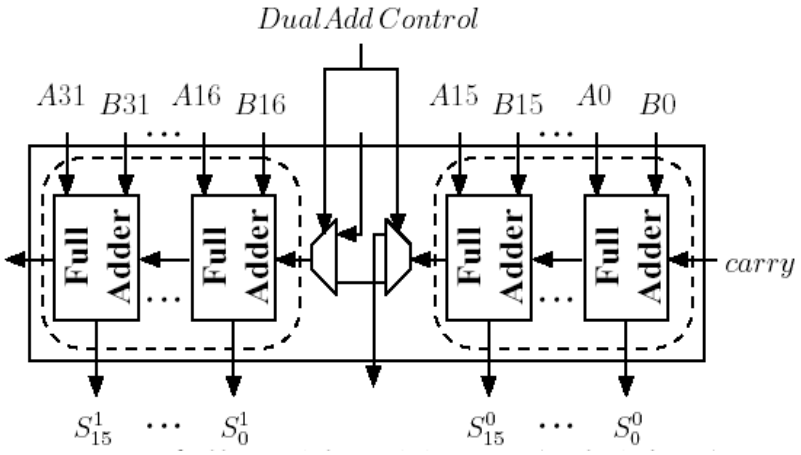


Figura 2: Un sumador de 32 bits convertido en un sumador SIMD de datos de 16 bits. Sólo es necesario añadir una señal de control y romper la cadena de acarreo incluyendo un multiplexor y un demultiplexor extra.

Además de las operaciones aritmético-lógicas más frecuentes, entre las que es frecuente incluir operaciones en aritmética con saturación[5], los repertorios se completaban con primitivas auxiliares y otras operaciones utilizadas con frecuencia en las aplicaciones multimedia. Aunque este tipo de operaciones son particulares de cada arquitectura y dependen de los tipos de datos y de los modos de acceso a memoria soportados, es habitual disponer de:

- Primitivas para la conversión de tipos y otras operaciones auxiliares para reorganización de los datos (empaquetado, desempaquetado, combinación y mezcla de componentes[6]).

- Instrucciones de desplazamiento, utilices para cómputos en aritmética con punto-fijo y para efectuar ciertas multiplicaciones y divisiones.

4 En algunos casos es posible efectuar de forma eficiente la multiplicación por medio de desplazamientos, sumas y restas. Uno de ellos es, por ejemplo, la multiplicación por constantes enteras o fraccionarias, muy frecuente en aplicaciones multimedia (filtrados) [20].

5 Con aritmética de saturación no es necesario manejar excepciones, que dificultan la respuesta en tiempo real.

6 Dependiendo del repertorio y del tipo de operación en concreto, se habla de permutaciones (*permutation*), mezclas (*mix* o *merge*), empaquetado y desempaquetado (*pack* y *unpack*), expansiones (*expansion*), conversiones (*conversion*), barajes (*suffle*)...

- Instrucciones de comparación para la generación de máscaras.

- Operaciones especiales de transferencia con memoria. En los repertorios de Intel se incluyen instrucciones para cargar datos no alineados. También son frecuentes instrucciones de acceso que no afectan a los contenidos de la jerarquía de memoria y stores selectivos (en función de una máscara).

1.2.4 Extensiones SIMD de Punto Flotante y Expectativas Futuras

Las primeras generaciones de extensiones SIMD sólo manejaban datos de tipo entero (o punto-fijo). A finales de los 90 sin embargo, comenzaron a añadirse nuevas extensiones SIMD que manejaban datos en punto flotante enfocadas al tratamiento de gráficos en 3D. El primero en incorporarlas fue AMD en 1998, con las extensiones 3DNow! en el procesador K6-2 [23]. Posteriormente llegaron las extensiones AltiVec de IBM-Motorola en el PowerPC G4 [24] y las sucesivas versiones de las SSE ("Streaming SIMD Extensions") de Intel [25]. Estas últimas ya utilizan un banco de registros específico para las nuevas extensiones, con registros más anchos de 128 bits. En las SSE2, se incluyeron operaciones en doble precisión [26]. Diferentes versiones con incorporación de nuevos repertorios de instrucciones se han ido incorporando en las sucesivas SSE3, SSSE3 y SSE4.

Las SSE3 se incorporan en el procesador Intel Prescott (PNI) a principios del 2004 [27], conocidas como la tercera generación para la arquitectura IA-32 (x86). En abril de 2005, AMD presentó un subconjunto de SSE3 en la revisión E de sus procesadores Athlon 64 (Venice y San Diego). SSE3 contiene 13 nuevas instrucciones, cuyo cambio más notable es la capacidad de trabajar horizontalmente en un registro, a diferencia de operaciones estrictamente verticales de las anteriores instrucciones SSE. Concretamente existen operaciones de suma y resta intercaladas (ADDSUBPS y ADDSUBPD) en las componentes de un registro. Estas instrucciones simplifican la implementación de una serie de operaciones comunes en procesadores de señal DSP y renderizado 3D. También incorporan instrucciones de conversión de datos de punto flotante a enteros sin tener que cambiar el modo de redondeo, evitando así costosas paradas en el pipeline.

SSE4 [28] consta de 54 nuevas instrucciones. Un subconjunto formado por 47 instrucciones, conocidas como SSE4.1 disponible en el procesador Penryn. Un segundo subgrupo formado por las siete instrucciones restantes conocido como SSE4.2 se ha incorporado en el primer procesador Core i7 con el nombre comercial de Nehalem. A diferencia de todas las anteriores SSE contiene instrucciones que ejecutan operaciones

específicas no multimedia (PTEST, MOVNTDQA). Existen varias instrucciones cuyo comportamiento está determinada por un campo constante y un conjunto de instrucciones (BLENDPS, BLENDPD, BLENDVPS, BLENDVPD, PBLENDVB, PBLENDW) que toman el registro XMM0 como tercer operando implícito. Varias de estas instrucciones permiten operaciones de baraje en un solo ciclo (INSERTPS, PINSRB, PINSRD/PINSRQ, EXTRACTPS, PEXTRB, PEXTRW, PEXTRD/PEXTRQ).
Los procesadores actuales de Intel Sundy Bridge y AMD con el Bulldozer incorporan el repertorio multimedia AVX (Advance Vector eXtension) siendo este una extensión a 256 bits de las anteriores SSE para x86.

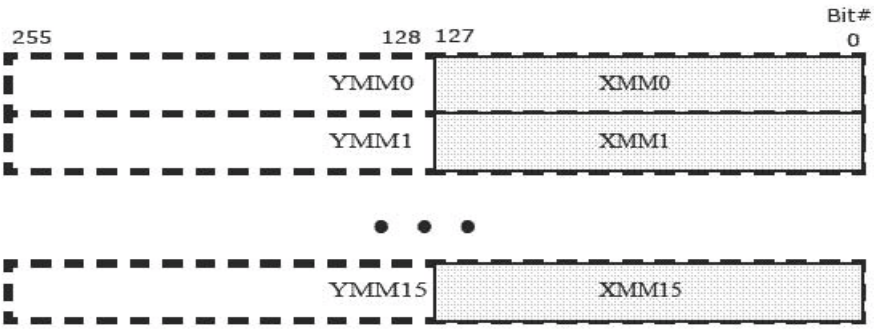


Figura 3: 16 registros vectoriales AVX de 256 bits extendiendo los XMM de 128 bits.

Sus principales diferencias con las anteriores se pueden resumir en:

- Incorporan soporte para 3 y 4 operandos no destructivos (en las SSE uno de los operandos fuentes operaba como destino).

- Amplían en ancho a 256 bits preparando el repertorio de instrucciones para futuros anchos superiores de 512 e incluso 1024 bits.

- Acceso alineado a memoria se encuentra relajado.

Y por último queríamos reseñar el repertorio multimedia de los procesadores Larrabee [29], un proyecto de Intel que trataba de incorporarse al mercado de hardware gráfico empleando arquitecturas x86.
Sin embargo las pobres perceptivas económicas y la competencia por un sector controlado ampliamente por fabricantes como nVIDIA, ha desembocado en una prematura cancelación del proyecto.

1.3 Necesidad de Simuladores en la docencia - Motivación

En los apartados anteriores se han ido abordando los retos tecnológicos y arquitectónicos que están presentes hoy en día a la hora de diseñar un nuevo microprocesador.

Si bien es cierto que la tecnología avanza a pasos agigantados, existe un recorrido importante en el conocimiento de Arquitectura y Tecnología de Computadores como queda visualmente descrito en la figura en 4. En ella se destacan algunos de los hitos arquitectónicos más importantes que se han ido incorporando a los procesadores de propósito general del fabricante Intel en el periodo 1988 hasta 2004 con el fin de dotarlos de mayores prestaciones.

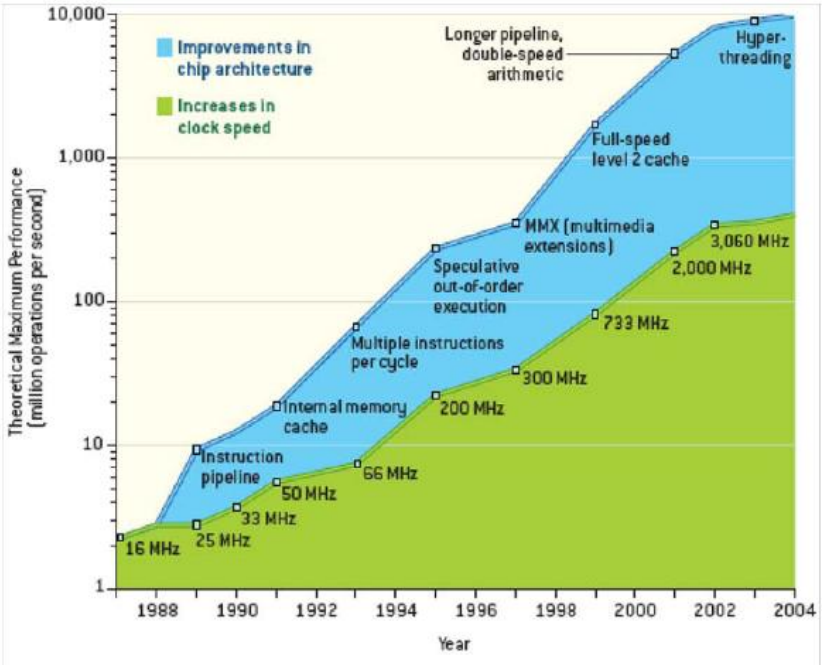


Figura 4: Mejoras tecnológicas y arquitectónicas incluidas los procesadores de Intel hasta el año 2004.

Gran parte de estas mejoras se abordan en diferentes asignaturas de la titulación de Ingeniero Superior en Informática [30] y el futuro grado [31] que se está implantando en la actualidad. No parece descabellado pensar en la necesidad docente de incorporar

herramientas didácticas basadas en simuladores con el fin de afianzar conceptos de dichas áreas de conocimiento.
Bajo esta premisa es encuadra este trabajo, dotar a los alumnos de una herramienta que permita visualizar el comportamiento de un procesador de propósito general al que se le incorporan mejoras arquitectónicas referentes al repertorio de instrucciones como son las extensiones multimedia.

1.3.1 Procesador segmentado DLX en ámbito docente

El DLX [1] [32] [33] es un microprocesador RISC diseñado por John Hennessy y David A. Patterson, los diseñadores principales de la arquitectura MIPS y Berkeley RISC, los dos ejemplos de la arquitectura RISC. Dicho procesador suele emplearse en ámbito académico para exponer los beneficios de un procesador segmentado y explicar su diseño. El capítulo 5 y 6 abordan la arquitectura de un procesador DLX, así como el repertorio de instrucciones y las características principales de dicha arquitectura.

Su propósito educativo ha derivado en simuladores gráficos de dicha arquitectura que permitan estudiar el comportamiento del repertorio de instrucciones, el cauce de instrucciones y el flujo de ejecución. En la literatura podemos encontrar varios simuladores específicos que basan su comportamiento en la arquitectura DLX:

- WinDLX [34]: uno de los más populares para entornos Windows desarrollado en la universidad de Viena. La aplicación permite visualizar en contenido de la memoria, registros y flujo de ejecución. Las etapas de las instrucciones se muestran con colores característicos y es factible seguir el flujo de ejecución.

- DLX Simulator (GNU GPL) [35]: desarrollado como proyecto fin de carrera en la Universidad de East Anglia de Norwich (Gran Bretaña). Emplea una consola para estudiar el comportamiento de las instrucciones. Destinado a entornos Unix/Linux puesto que incorpora un fichero Makefile así como el código fuente.

- ESCAPE [36]: desarollado por Peter Verplaetse y Jan Van Campenhout en el departamento de Electrónica y Sistemas Informáticos de la Universidad de Gante (Bélgica). Destinado a entornos Windows permite no solo seguir el flujo de ejecución y visualizar los contenidos del banco de registros y memoria, sino que es capaz de mostrar detalladamente la información (en binario) del camino de datos y de las unidades de control.

- DLXview [37]: desarrollado por la universidad de Padua (Italia). Creado como extensión gráfica del DLXSim (DLX no gráfico). El modelo simple de pipeline de DLXSim se ha mejorado para soportar los modelos de ejecución scoreboarding, y el algoritmo de Tomasulo...

No obstante, encontramos ejemplos de variantes del procesador DLX que incorporan algunas novedades arquitectónicas en ámbitos docentes, como el DLXV (procesador vectorial[7] descrito por Hennessy & Patterson en [1]) y el VLIW-DLX para procesadores VLIW [38] desarrollado por la Universidad de Praga.

Sin embargo, este tipo de simuladores no llegan a incorporar evoluciones en el repertorio de instrucciones, como las extensiones multimedia que están presentes en todos los procesadores de propósito general hoy en día. Este motivo ha llevado la puesta en marcha de este trabajo encuadrado en la asignatura de Proyecto de Sistemas Informáticos. La idea es incorporar las prestaciones de algún repertorio multimedia en un procesador que sea caso de estudio habitual en el área de conocimiento de Arquitectura y Tecnología de Computadores.
Por este motivo nos hemos decantado en crear un simulador de un procesador DLX que se le incorporen AltiVec [24] de IBM-Motorola en el PowerPC G4. La elección del repertorio AltiVec viene motivada porque el procesador PowerPC G4 es un procesador RISC al igual que el DLX del ámbito docente.

1.4 Desarrollo de la memoria

En este trabajo de Sistemas Informáticos se ha desarrollado una aplicación multiplataforma que simule el comportamiento de un procesador DLX al que se le han incorporado las extensiones multimedia AltiVec que hemos bautizado como DASIT (siglas de “DLX-AltiVec Simulation Tool”).

Esta memoria ha quedado dividida en varios capítulos de forma que el capítulo 2 describa el procesador DLX, el capítulo 3 el repertorio multimedia AltiVec, el capítulo 4 nuestro trabajo desarrollado en el simulador DASIT, y capítulo 5 una guía de uso del simulador, el capítulo 6 constará de un caso ejemplo, y por último el capítulo 7 abordará las principales conclusiones de este trabajo.

7 DLXV ha sido desarrollado por el Departamento de Ingeniería de Sistemas y automática de la Universidad Politécnica de Valencia y puede encontrarse en el portal web: http://informatica.uv.es/docencia/software/dlxv.htm

Capítulo 2

Procesador DLX

Índice:

2.1 Introducción

El DLX es un microprocesador de tipo RISC, diseñado por John Hennessy (diseñador principal de la arquitectura MIPS) y David A. Patterson (diseñador de la arquitectura Berkeley RISC).

Se podría considerar el DLX como un MIPS revisado y más simple con una arquitectura simplificada de carga y almacenamiento de 32 bits. Está diseñado esencialmente para propósitos educativos, ya que se utiliza ampliamente en cursos universitarios para los temarios de arquitectura de computadores.

2.2 Arquitectura tipo RISC

La arquitectura computacional RISC (del inglés “Reduced Instruction Set Computer”) es un tipo de microprocesador que tiene principalmente las siguientes características:

- Las instrucciones son de tamaño fijo y están presentadas en un reducido número de formatos.
- Sólo tienen acceso a la memoria de datos las instrucciones de almacenamiento y carga.

El objetivo principal de diseñar máquinas utilizando esta arquitectura, se basa en posibilitar la segmentación y el paralelismo en la ejecución de instrucciones, así como en reducir los accesos a memoria.
La relativa sencillez de esta arquitectura, lleva a ciclos de diseño más cortos cuando se van desarrollando nuevas versiones, lo que posibilita que se pueda hacer una buena aplicación de las nuevas tecnologías de semiconductores. Por ello, los procesadores con arquitectura RISC, además de tender a ofrecer una capacidad de procesamiento del sistema de 2 a 4 veces mayor, los saltos de capacidad que se producen de una a otra generación de procesadores son mucho mayores que en los CISC (del inglés "Complex Instruction Set Computing").
La idea de crear una arquitectura RISC, fue inspirada por el hecho de que casi todas las características que se iban incluyendo en los diseños tradicionales de CPU con la intención de aumentar la velocidad, estaban siendo ignoradas por los programas que se ejecutaban en ellas. Además, la velocidad del procesador cuando debía interactuar con la memoria de la computadora, era cada vez más alta. Esto llevó a la aparición de múltiples técnicas para intentar reducir el procesamiento dentro del CPU, y a su vez intentar reducir el número de accesos a memoria total.

Los diseños RISC han llevado al éxito, a muchas plataformas y arquitecturas, algunas de las más importantes:

- Microprocesadores MIPS, integrados en la mayoría de las computadoras de Silicon Graphics hasta 2006, y en consolas ya descatalogadas, como Nintendo 64, PlayStation y PlayStation 2. Actualmente se utiliza en la PlayStation Portable y algunos routers.

- La serie IBM POWER, utilizado en Servidores y superordenadores por IBM.

- La versión PowerPC de Motorola e IBM, utilizada en los ordenadores AmigaOne, Apple Macintosh como el iMac, eMac, Power Mac y posteriores (hasta 2006). Actualmente se utiliza en muchas consolas, como la Playstation 3, Xbox 360 y Nintendo Wii.

- El procesador SPARC y UltraSPARC de Sun Microsystems y Fujitsu, que se puede encontrar en sus últimos modelos de servidores.

- El PA-RISC y el HP/PA de Hewlett-Packard, ya descatalogados por completo.

- El DEC Alpha en servidores HP AlphaServer y estaciones de trabajo AlphaStation, ya descatalogados.

- Los procesadores ARM, que dominan en PALM, Nintendo DS, Game Boy Advance y en múltiples PDAs, Apple iPods, Apple iPhone, iPod Touch, Apple iPad, y videoconsolas como Nintendo DS, Nintendo Game Boy Advance.

- El Atmel AVR, usado en muchos productos, desde mandos de la Xbox a los coches de la empresa BMW.

- La plataforma SuperH de Hitachi, originalmente usada para las consolas Sega Super 32X, Saturn y Dreamcast. Ahora forman parte de muchos equipos electrónicos para el consumo.

- Los procesadores XAP, utilizados en muchos chips wireless de CSR (Bluetooth, wifi).

Nos centraremos en el microprocesador MIPS, ya que el DLX está inspirado en él.

2.3 El microprocesador MIPS

El DLX está inspirado en particular en el microprocesador MIPS (siglas de "Microprocessor without Interlocked Pipeline Stages"). Con este nombre se identifica a toda una familia de microprocesadores de arquitectura RISC desarrollados por "MIPS Technologies", una compañía que se dedica a desarrollar microprocesadores, que fue fundada en 1984 bajo el nombre de "MIPS Computer Systems Inc." por el Dr. John Hennessy de la Universidad de Stamford, quien dirigió el proyecto "MIPS RISC architecture" desde 1981. Fue pionera en la producción de procesadores RISC

Los diseños del MIPS son usados para productos informáticos de SGI ("Silicon Graphics International"), en muchos sistemas embebidos, en dispositivos para Windows CE, routers Cisco, y videoconsolas como la Nintendo 64 o las diferentes consolas de Sony, como la PlayStation, PlayStation 2 y PlayStation Portable, aunque la ultima consola de esta familia, la PlayStation 3, no utiliza un MIPS, sino que utiliza un PowerPC.

Las primeras arquitecturas MIPS, estaban implementadas en 32 bits (tanto para las rutas de datos, como para los registros de 32 bits de ancho), pero las versiones posteriores fueron implementadas en 64 bits. Actualmente, existen cinco revisiones compatibles con el conjunto de instrucciones del MIPS, llamadas "MIPS I", "MIPS II", "MIPS III", "MIPS IV" y "MIPS 32/64". En la última de ellas, la "MIPS 32/64 Release 2", se define mucho mejor el conjunto de control de registros. Además, se pueden encontrar disponibles varias extensiones:

- MIPS-3D, que consiste en un simple conjunto de instrucciones SIMD en coma flotante dedicadas a tareas 3D.
- MDMX (MaDMaX), que está compuesta por un conjunto mucho más extenso de instrucciones SIMD enteras, las cuales utilizan los registros de coma flotante de 64 bits.
- MIPS16, que comprime el flujo de instrucciones para conseguir que los programas ocupen menos espacio (supuestamente, como respuesta a la tecnología de compresión Thumb de la arquitectura ARM).
- La reciente MIPS MT, que añade funcionalidades multithreading de grano fino.

Gracias a que los diseñadores del microprocesador MIPS, crearon un conjunto de instrucciones tan claro, los cursos en universidades y escuelas técnicas, sobre el tema de arquitectura de computadores, normalmente se basan en la arquitectura MIPS.

El diseño de la familia de CPU's MIPS influiría de manera importante en otras arquitecturas RISC posteriores como los DEC Alpha.

2.4 Instrucciones DLX

Las instrucciones DLX se pueden separar en tres tipos según su comportamiento: tipo R, tipo I y tipo J.

Las instrucciones de tipo R, son instrucciones de registro puras. Están formadas por una palabra de 32 bits que representa, un operando y tres registros.

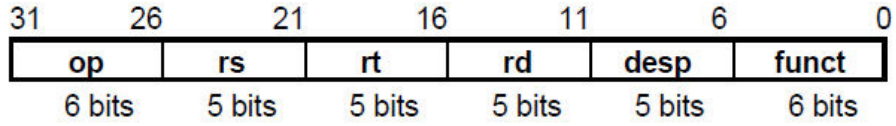


Figura 5: Formato de Instrucción de Tipo R.

Las instrucciones de tipo I son similares, pero sólo incluyen un registro, y usan los otros 16bits (los que en las instrucciones de tipo R sirven para indicar los otros dos registros), para almacenar valores inmediatos.

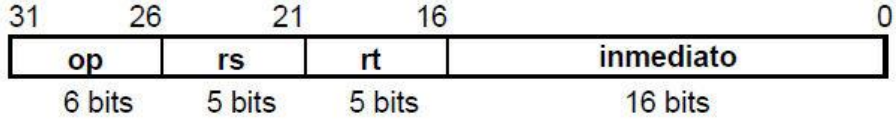


Figura 6: Formato de Instrucción de Tipo I.

Las instrucciones de tipo J son saltos, formados por un operando y una dirección de salto de 26 bits.

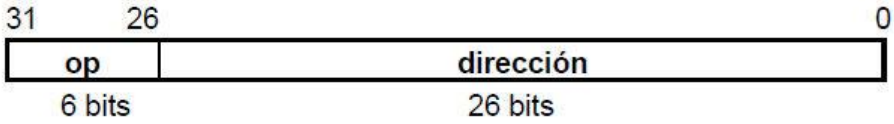


Figura 7: Formato de Instrucción de Tipo J.

Los códigos de operación (también conocidos como “opcodes”) tienen una longitud de 6 bits, con lo que podría haber un total de 64 posibles instrucciones básicas. Se necesitan 5 bits para seleccionar cada uno de los 32 registros. En algunas instrucciones no todos los bits son usados, por ejemplo, en el caso de las instrucciones de tipo J, sólo 18 de los 32 bits de la palabra son utilizados, lo que implica que los 6 bits menos significativos, se puedan aprovechar para indicar “instrucciones extendidas”. Esto permite que el DLX pueda tener más de 64 instrucciones, siempre y cuando éstas sólo trabajen con registros.

2.5 Segmentación

El DLX, al igual que el microprocesador MIPS, basa su rendimiento en el uso de la segmentación.

La segmentación consiste fundamentalmente, en descomponer la ejecución de cada instrucción, en varias etapas, de tal manera que cada una de ellas tarde un ciclo de reloj. De esta manera se puede empezar a procesar una instrucción nueva cada ciclo y trabajar con varias instrucciones a la vez, de manera que nunca dos instrucciones compartan una misma etapa.

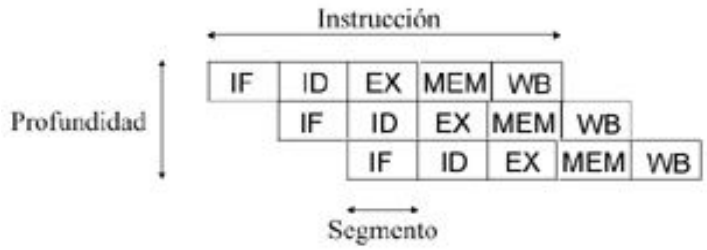


Figura 8: Pipeline Segmentado de un procesador DLX.

Cada una de las etapas de una instrucción, usa un hardware determinado dentro del procesador, de forma que la ejecución de cada etapa, no interfiere en la ejecución del resto.

Si no existiera segmentación en el procesador, no se podría ejecutar la siguiente instrucción hasta la finalización de la anterior. Sin embargo, con un procesador segmentado, exceptuando casos de dependencias de datos o uso de unidades funcionales, la siguiente instrucción puede iniciar su ejecución justo después de acabar la primera etapa de la instrucción actual.

La segmentación del microprocesador DLX tiene cinco etapas:

- IF: (Búsqueda) Esta etapa es la encargada de la obtención de la instrucción dentro de la memoria de instrucciones.

- ID: (Decodificación) Etapa en la que se decodifica la instrucción. Esta unidad toma la instrucción dada en la etapa IF, y extrae el código de operación y los operandos, obteniendo los valores necesarios del banco de registros.

- EX: (Ejecución de la unidad aritmético-lógica) Ejecuta la instrucción, ya sea usando la Unidad Aritmético-Lógica (siglas en inglés “ALU”) entera o en coma flotante. Se considera que la Unidad Funcional entera (utilizada para operaciones enteras) dura un ciclo de reloj, sin embargo, las distintas unidades funcionales en coma flotante pueden tener un número de ciclos variable. Lo que hace que la etapa EX no dure siempre un ciclo de reloj. Por ello, y para evitar que el pipeline se quede bloqueado, actualmente se utilizan unidades funcionales segmentadas.

- MEM: (Memoria) En esta etapa se accede a la Memoria de Datos. Las únicas instrucciones que acceden son las de carga y almacenamiento. Las instrucciones de carga, buscan la dirección de memoria y obtienen los datos que necesitan. Sin embargo las de almacenamiento, almacenan el dato en la dirección de memoria calculada en la etapa de ejecución correspondiente.

- WB: (ReEscritura, del inglés “WriteBack”) Esta etapa es la encargada de guardar en el banco de registros la solución de la operación.

2.6 Riesgos de la segmentación

Los riesgos, se llaman a las situaciones que impiden que se ejecute la siguiente instrucción durante su ciclo determinado. Se introduce un ciclo de parada.

Existen tres tipos de riesgos:

Riesgos estructurales: Son aquellos riesgos que se dan cuando al existir múltiples instrucciones en el pipeline las mismas etapas se solapan y tratan de acceder a la misma unidad funcional.

Riesgos por dependencias de datos: Son los riesgos ocasionados por conflictos de dependencias de datos entre distintas instrucciones que se encuentran en el pipeline al mismo tiempo.

Riesgos de Control: Principalmente se refiere a los problemas que surgen cuando no se conoce la siguiente instrucción correcta a ejecutar.

Capítulo 3

Extensiones Multimedia ALTIVEC

Índice:

3.1 Introducción

AltiVec representa un conjunto de instrucciones que fueron diseñadas y que actualmente están en propiedad de IBM, Motorola y Apple Computer, que se unieron para formar el nuevo conjunto AltiVec, basado en la arquitectura del PowerPC. A partir de las versiones G4 de motorola y G5 de IBM se añadió a la arquitectura el conjunto de AltiVec.

AltiVec es hoy en día, una marca registrada que pertenece a Motorola. Por ello si se desea obtener más información se puede buscar como “Velocity Engine” nombre que registro “Apple” y como “VMX” referenciado así por IBM.

Desde ahora nos referiremos bajo el nombre de “AltiVec” para generalizar.

EL conjunto AltiVec fue incorporado en ordenadores de sobremesa a finales de años 1990, tras haber sido desarrollado entre 1996 y 1998 por la colaboración de las tres compañías ya mencionadas. AltiVec fue entonces el sistema SIMD (del inglés “Single Instruction, Multiple Data”) más potente en la CPU.

Según la taxonomía de Flynn, los repertorios SIMD implican que en una sola instrucción se aplica la misma operación a varios datos, nos referimos entonces a paralelismo a nivel de datos.

En el momento de su aparición, AltiVec se encontraba a la cabeza de sus contemporáneos similares. Respecto a MMX de Intel, la cual operaba con elementos enteros, y a SSE con elementos en coma flotante, AltiVec ponía a disposición un mayor número de registros a emplear, con un mayor número de formas posibles respecto a sus competidores y usando un conjunto de instrucciones más flexible. Convirtiéndose por el momento en un conjunto inigualable hasta la incorporación de SSE2 en los Pentium 4.

3.2 Estructura AltiVec

En cuanto a las características particulares de las que dispone AltiVec, se halla un conjunto de instrucciones que permiten el control de la cache para mejorar el funcionamiento al trabajar con un mayor flujo de datos. Además AltiVec soporta datos especiales como pixel de RGB. Sin embargo AltiVec tiene sus desventajas, al no permitir operaciones de punto flotante de doble precisión (64 bits) ni tampoco dispone de instrucciones para desplazar o copiar los datos entre los registros directamente sino que deben realizarse mediante cargado y almacenamiento en memoria. Siguiendo de esta manera el modelo del diseño RISC del PowerPC que solo puede cargar y almacenar desde la memoria.

La clave del triunfo que resulto AltiVec, fue que sus registros disponen de un ancho de 128 bits, que comparado con la arquitectura DLX de 32 bits, supone una mejora de cuatro veces el espacio para los datos e instrucciones. A razón de dicha mejora AltiVec dispone de operaciones que permiten el uso de hasta cuatro operandos en la misma instrucción, siendo 3 de tipo fuente y un operando destino. Paralelamente a nivel de instrucción al disponer de un tamaño de 128 bits, AltiVec permite realizar la misma operación sobre varios datos a la vez, contenidos en esos 128 bits.

El repertorio dispuesto por AltiVec, permite pues realizar cuatro operaciones de tamaño de palabra, 32 bits, con una sola instrucción. Sin embargo, no es ésta la única mejora que presenta, además el repertorio permite también realizar operaciones sobre tamaños de datos de 8 bits y 16 bits dando lugar a 16 operaciones y 8 operaciones respectivamente por instrucción ampliando así las posibilidades de operaciones existentes.

Los datos en AltiVec de memoria siempre están alineados en 16 bytes.

El conjunto de instrucciones AltiVec permite que sobre los datos mencionados anteriormente se tenga en cuenta o no el signo de los datos contenidos en el vector. De esta manera AltiVec es capaz de operar con elementos de tipo carácter con y sin signo, elementos enteros tipo short con y sin signo, con enteros y además con flotantes de simple precisión. Dejando de lado los flotantes de doble precisión.

En versiones recientes de compiladores (GCC, Visual Age IBM) se puede acceder a las instrucciones AltiVec desde programas en C y C++, indicando los datos como "vector" y mediante las funciones como por ejemplo "vec_add" (listadas en el capítulo "Instrucciones Multimedia")

El tipo de datos con los que se puede operar con el repertorio AltiVec es bastante extenso:

- vector unsigned char
- vector signed char
- vector bool char
- vector unsigned short
- vector signed short
- vector bool short
- vector pixel
- vector unsigned int
- vector signed int
- vector bool int
- vector float

Por ejemplo, al declarar "vector unsigned char v1", v1 deberá formarse con 16 elementos de tipo carácter sin signo.

3.3 Aplicaciones AltiVec

La mayoría de las aplicaciones de imagen y sonido no requieren más de 8 o 16 bits de datos para representar el color y el sonido. AltiVec puede ayudar al procesamiento de las siguientes aplicaciones:

- Voz sobre IP (VoIP). Es un grupo de recursos que hacen posible que la señal de voz viaje a través de Internet.

- Concentradores de acceso / DSLAM. Un concentrador de acceso toma los datos de un Pots(Servicio telefónico Ordinario Antiguo) y los introduce en Internet. El

DSLAM (Multiplexor de acceso a la línea digital de abonado) proporciona a los abonados acceso ADSL.

- Reconocimiento de Voz. El reconocimiento de voz permite el procesamiento de la voz para su uso en aplicaciones como asistendia de directorio y marcación automática.

- Procesamiento de sonido (Audio de codificación y decodificación). El procesamiento de voz utiliza la señal para mejorar la calidad del sonido en las líneas.

- Comunicaciones:
    - Multicanal de los módems.
    - Los registros de módems pueden utilizar la tecnología AltiVec para sustituir a los procesadores de señal DSP(Procesador digital de señales).

- Gráficos 2D y 3D: Juegos arcade.

- Procesamiento de imagen y video: JPEG, filtros.

- Cancelación del eco. Se utiliza para eliminar el eco en las llamadas con retardo (250-500 milisegundos, por ejemplo comunicaciones por satélite).

- Estación base de ejecución. Comprime los datos digitales de voz para la transmisión a través de Internet.

- Videoconferencia.

3.4 AltiVec en la Actualidad

Actualmente Apple es considerado el principal usuario de AltiVec. Lo emplea en sus aplicaciones multimedia (QuickTime o iTunes) para acelerar la reproducción y rendimiento. Además lo aplica a programas de procesamiento de imágenes como Adobe Photoshop. Por último, en lo referente a Apple, AltiVec aparece en su sistema operativo “Mac OS X“, en el apartado del compositor de gráficos Quartz.

Desde su “G4”, Motorola ha incorporado AltiVec en todos sus dispositivos PC de sobremesa.

Por su parte, IBM ha eliminado a VMX de sus sistemas microprocesadores “POWER“, dado que en su mayoría se emplean para aplicaciones de servidor donde no es requerido el repertorio multimedia AltiVec. Aunque de IBM, en el microprocesador “PowerPC 970” (G5 de Apple), se encuentra una unidad AltiVec de alto rendimiento, la cual está compuesta por dos unidades funcionales para permitir operaciones superescalares. Un VMX completo en una unidad, y un multiplicador/sumador en la otra.

Este microprocesador lo podemos encontrar en el superordenador más potente de España, el “MareNostrum”, que se encuentra en Barcelona y actualmente ocupa la posición 170 dentro de la lista de superordenadores.

Capítulo 4

DLX-AltiVec Simulation Tool

Índice:

"DASIT" (siglas de "DLX-AltiVec Simulation Tool") consiste en un simulador de un procesador DLX combinado con un repertorio de instrucciones multimedia en concreto el conjunto de instrucciones AltiVec.

Figura 9: Icono de inicio del DLX-AltiVec Simulation Tool.

"DASIT" supone una novedad en las herramientas didácticas orientadas a la arquitectura de computadores. Principalmente es el único que permite la compilación de instrucciones del repertorio AltiVec bajo un comportamiento de arquitectura DLX. Para dar forma a este proyecto, se opto por emplear el lenguaje de programación java, al margen de por su facilidad y manejabilidad a la hora de programar permite que la herramienta sea en principio portable a cualquier sistema operativo que posea la máquina de java.

Un simulador DLX y en concreto "DASIT" se compone principalmente de 3 elementos. El código que se desea ejecutar, el pipeline que muestra el estado de la ejecución y los bancos de registros y la memoria de datos e instrucciones, en este caso solo de instrucciones para comprobar el estado de los datos que se están manejando. Dichos componentes forman el simulador "DASIT" descrito a continuación.

Además se han añadido funcionalidades como la visualización de las estadísticas para un mejor aprendizaje del comportamiento de programas con instrucciones DLX o AltiVec. Las opciones de guardar y cargar códigos ya existentes con extensión .s. O la posibilidad de modificar cualquier valor de la memoria de datos o del banco de registros manualmente en cualquier momento.

Una pieza clave que no se encuentra en otros simuladores es la posibilidad, de visualizar un gráfico con el estado de la ruta de datos donde se puede apreciar por donde están discurriendo los datos en ese estado del pipeline.
Todas las funcionalidades descritas pueden llevarse a cabo fácilmente mediante la interfaz de usuario intuitiva implementada en "DASIT". En caso de dificultades siempre se podrá acceder al manual de usuario y los repertorios de instrucciones mediante la ayuda.

A nivel de código el proyecto está fragmentado en cuatro conjuntos distintos para distribuir el código según su función como se muestra en el diagrama de clases siguiente. Los 4 conjuntos en los que se encuentra repartido el código coinciden parcialmente con los elementos principales que comentamos antes que requiere un simulador DLX, el compilador o parser, que recupera el código introducido por el usuario y lo prepara para ejecución. El conjunto de componentes que posee entre sus clases el banco de registros, la memoria principal y la unidad de detección de riesgos. El paquete funcionamiento responsable de enlazar todos las partes y que incluye las clases que controlan el comportamiento y visualización del pipeline. Y por último el conjunto que implementa las instrucciones de ambos repertorios, DLX y AltiVec para las 5 etapas características de la segmentación DLX.

Distribución de las clases implementadas:

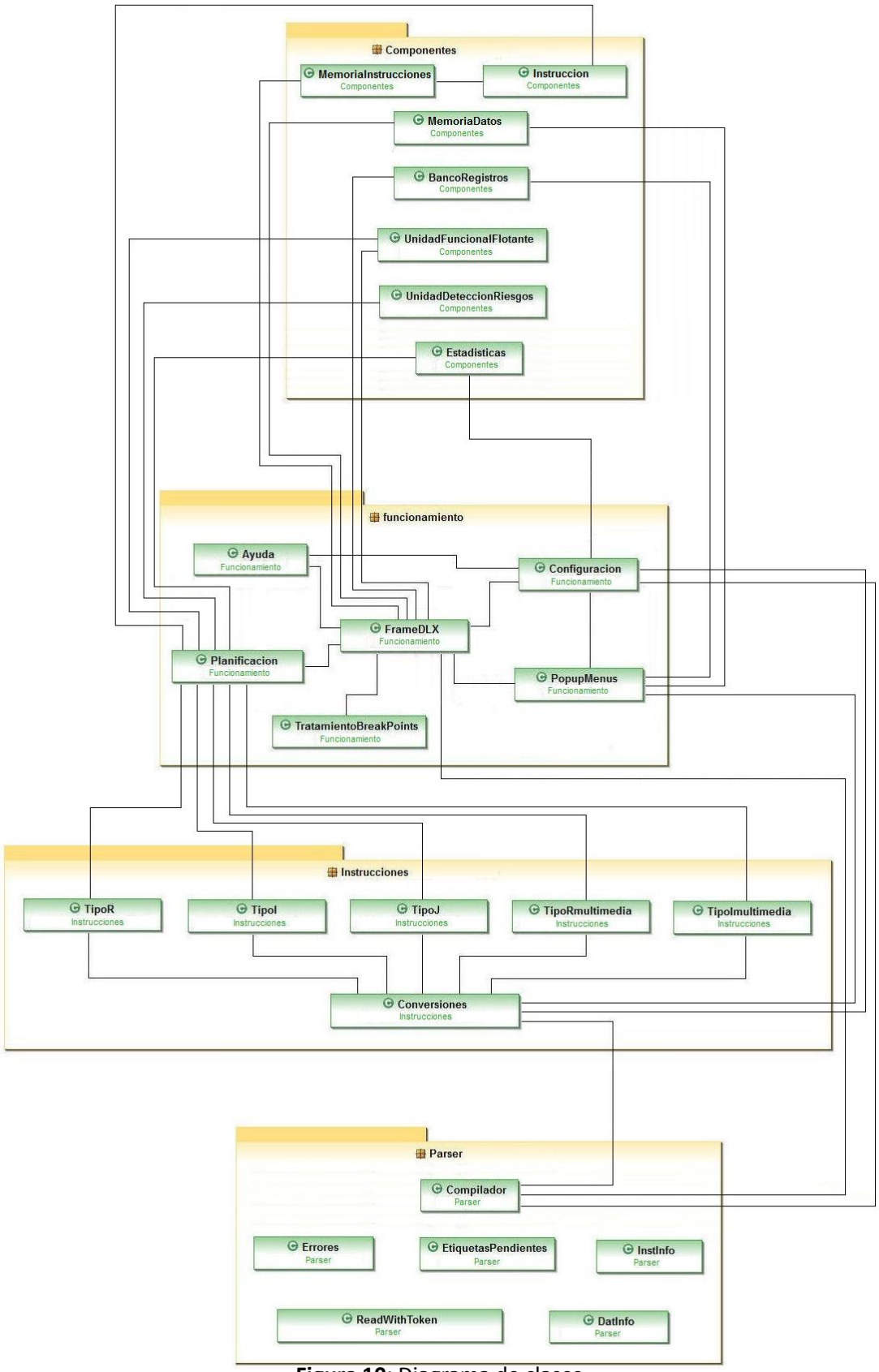


Figura 10: Diagrama de clases.

4.1 Módulo Componentes

Índice:

4.1.1 Instrucción

La clase “Instrucción” contenida en el proyecto es probablemente la más importante que podremos encontrar, el motivo de semejante afirmación es porque cada instancia de esta clase representa una de las instrucciones que se han compilado del código recuperado en el Parser.
Aunque en teoría las instrucciones de un procesador DLX son de tipo RISC y no deberían ser complejas, la clase Instrucción contiene una cantidad considerable de atributos. Comenzaremos por justificar semejante cantidad, por el hecho de que las instancias de Instrucción en verdad también representan los buffers existentes en la segmentación DLX donde se guardan los datos de cada etapa, y las variables de control.

Tipo de Instrucción, operación y operandos

Un primer grupo de elementos que componen las instancias son la información referente a la instrucción. Por un lado estos elementos son el código de operación de la instrucción para distinguir todas las instrucciones de las que disponemos en los repertorios DLX y AltiVec. Y por otro lado los registros destino y los registros fuente que serán accedidos en la ejecución de la instrucción.
Dado que el planificador necesita identificar el tipo de registro es necesario conservar el valor del registro accedido dentro del rango 0 a 31 y el tipo de registro accedido, R enteros, F flotantes o M multimedia. Dichos atributos resultan imprescindibles en el planificador, principalmente para comprobar la necesidad o no de anticipación y la dependencia de valores. Por ello el planificador y la unidad de detección de riesgos requieren el acceso a estos datos.

Para que el planificador identifique correctamente el comportamiento a seguir con la instrucción, la instancia de “Instrucción” llevara el tipo al que pertenece la instrucción. (Ver tipos de instrucciones, en los apéndices I y III).

Figura 11: Clase Instrucción.

Valor de los Operandos en función del tipo

Un segundo conjunto de atributos es el resultado de que en el simulador el tipo de datos que se maneja en la instrucción depende del tipo de instrucción, en concreto se refiere a si la instrucción emplea enteros representados como el tipo int de java, flotantes representados como el tipo, o datos multimedia que se representan en cadenas de strings de tamaño 128 y conteniendo ’1’s y ‘0’s.
El porqué del empleo de estos tipos de datos y no de una generalización binaria, es el resultado de la simplicidad que ofrece java a la hora de operar con los enteros y flotantes directamente evitando tediosas conversiones y operaciones en binario.

Por ello “Instrucción” dispone de 3 campos para almacenar la información de datos en enteros, lo correspondiente al dato del registro destino, y los dos datos que podría haber como fuentes. Además incluye 4 campos para los datos multimedia, un dato que irá al destino y 3 posibles datos fuente. Por último los campos para los datos flotantes, estos requieren una explicación más detallada.
Dado que las operaciones del repertorio DLX pueden operar con dobles, emplearemos 6 campos, se debe a que los datos contenidos en “Instrucción” en el caso de una operación flotante si es en simple precisión almacena un dato destino y dos datos fuente, pero al emplear la doble precisión es necesario un campo extra por cada campo dado que los datos se almacenan en float.
A razón de esto cada parte del flotante en doble precisión se almacena, su primera parte codificada a flotante en el campo empleado por los flotantes de simple precisión y la segunda parte codificada igualmente en flotante en los campos extra identificados con el sufijo “Double”. En el momento de operar con los flotantes de doble precisión se tomaran ambos flotantes y se juntaran mediante conversiones para obtener el elemento deseado.

Distinción entre simple y doble precisión

El conflicto entre simple y doble precisión da lugar a tener que distinguir en varios puntos del código, en el pipeline del planificador sobre todo, si se trata de simple o doble precisión dado que en función de ello si se trata de un double, se debe recuperar el segundo registro es decir el registro indexado +1 y almacenar en el registro indexado +1 el segundo elemento que forma el flotante de doble precisión y realizar las conversiones pertinentes. Por ello la instancia de Instrucción incluye un atributo booleano que identifica si se trata o no de una instrucción doble.

Inmediato

Dentro de la necesidad de almacenar datos existe un campo reservado para el inmediato, empleado en las instrucciones de salto y de accesos a memoria u operaciones con inmediato del repertorio DLX.

Carga y almacenamiento en memoria

En relación al acceso a memoria destacar la necesidad de un campo extra solo empleado para el almacenamiento de datos, es decir el dato que se desea almacenar, el resultado se guarda en el atributo de la instancia en forma de cadena de caracteres de tamaño 32, correspondiente al formato de la memoria. Semejante a este campo existe uno similar que soporta 8bits y 16bits reservado para el almacenamiento de halfwords o bytes en la memoria, dado que difieren de tamaño de los datos estándar.
Por último en cuanto a los datos para las operaciones de las instrucciones se encuentra un espacio para la dirección de memoria a la que se desea acceder y en caso de que sea un flotante de doble precisión el dato cargado o almacenado existe un campo con la dirección de su segunda parte.

Variables de Control

Como ya se comento al inicio de esta explicación una instancia de "Instrucción" representa también el estado del buffers de cada etapa en función de la etapa en que se encuentre la instrucción.
Por este motivo es necesaria la presencia de las variables de control dentro de la Instrucción.

Todas estas variables serán consultadas y modificadas por la unidad de detección de riesgos y el planificador. Entre ellas encontramos la información sobre si se da o no la saturación para las instrucciones multimedia para la activación del bit de saturación del registro especial multimedia.
También se halla una variable para describir si dicha operación debe o no ejecutarse, esta variable viene a razón del salto, dado que si se toma, la siguiente instrucción queda anulada por el planificador.

Y las últimas variables de control que caben destacar serían las que informan de si se ha realizado ya la primera ejecución, en el caso de que la instrucción disponga de varias etapas de ejecución. De esta manera podremos distinguir entre los dos siguientes casos:

- Una instrucción entra, desde su etapa ID, a la etapa EX parada. Por lo que no consume ningún ciclo de ejecución, y debemos marcar que aun no hemos realizado la primera ejecución.

- Una instrucción hace su etapa EX 1 y justo después pasa a la etapa EX parada. Ocurre pues, que no se ha consumido ningún ciclo de ejecución, puesto que este se gasta cuando pasemos a la siguiente etapa de ejecución. Por lo que la única manera de distinguir este caso con el anterior, es marcar que en este caso sí que hemos realizado la primera ejecución.

Identificación de la etapa

Finalmente no podría faltar la presencia de los estados en los que se encuentra una instrucción en forma de etapa y de etapa anterior. En este punto se recuerda que la etapa en la que se encuentra es la etapa que va a llevarse a cabo cuando el pipeline tome el control de esa ejecución y por lo tanto etapa anterior representa la etapa que se muestra en la tabla que simula el pipeline (más detallado en el planificador)

Uso y funciones

En cuanto al comportamiento de esta clase únicamente es una clase contenedor, las escasas acciones que realiza son actualizar su siguiente etapa en función de su etapa actual y de si existen o no paradas, determinadas por la unidad de control de riesgos.

4.1.2 Banco de Registros

El banco de registros esta simulado mediante vectores de tamaño 32 bytes. Como consideramos que el banco de registros está dividido en tres partes, enteros, flotantes y multimedia, necesitamos un vector por cada tipo de registro.

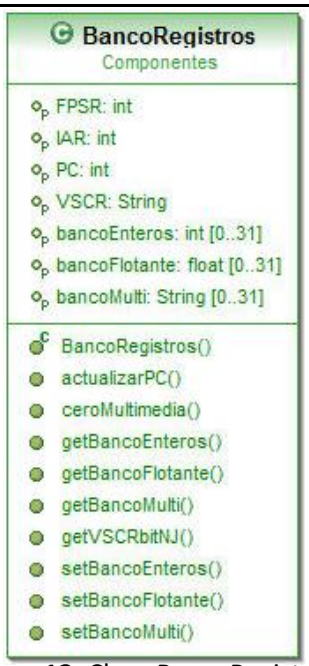


Figura 12: Clase BancoRegistros.

Para los 32 registros enteros usaremos un vector de enteros, para los flotantes, utilizaremos un vector de números flotantes y para representar los registros

multimedia, necesitaremos un vector de cadena de caracteres, concretamente de 128 bytes, ya que guardaremos los datos en codificación binaria.
Acorde con la representación de los datos que contiene el simulador DLX, aparte de los registros ya mencionados, existen otros campos para referenciar los registros especiales.

Contamos con un registro general, llamado PC (del inglés "Program Counter") o contador de programa, que esta añadido en todas las zonas del Banco de registros.

Además contamos con un registro especial por cada uno de los tipos de registros que soporta nuestro Banco de Registros:

- El Registro entero IAR (del inglés "Interrupt Adress Register"), que representa el registro para la dirección de interrupción.
- El Registro flotante FPSP (del inglés "Float-Point Status Register"), que representa el registro de estado de punto flotante.
- El registro multimedia VSCR, que representa el registro multimedia para indicar la saturación o el modo non-java.

4.1.3 Memoria de Instrucciones

Disponemos de una instancia de la clase MemoriaInstrucciones, donde se encuentra el listado completo de todas las instrucciones que han sido compiladas, y que pasaremos como parámetro al planificador para que vaya extrayendo la que corresponde en cada ciclo de reloj. El listado es un vector de elementos de tipo "Instrucción" cada una representando una instrucción. Representa la memoria de instrucciones.

Figura 13: Clase MemoriaInstrucciones.

4.1.4 Memoria de Datos

La memoria, consiste exclusivamente en una memoria de datos, ya que la memoria de instrucciones, no es visible al usuario.

En el simulador está representada por un vector de cadenas de caracteres. El tamaño de la memoria del simulador es de 2048 elementos, lo que implica 2048*4 bytes de memoria a las que se puede acceder. De esta manera la memoria solo puede ser referenciada mediante 13 bits, 2^{13} posiciones de memoria.

Figura 14: Clase MemoriaDatos.

4.1.5 Unidad Funcional Flotante

La instancia que se emplea en el proyecto de esta clase tiene como finalidad servir a la unidad de control de riesgos, mediante la configuración el usuario podrá modificarla, la unidad se inicializa con los valores existentes en el momento en que el usuario compila el código contenido en el campo de texto. La unidad funcional es necesaria dado que informara del estado de las 4 unidades funcionales flotantes de las que dispone el simulador DLX.

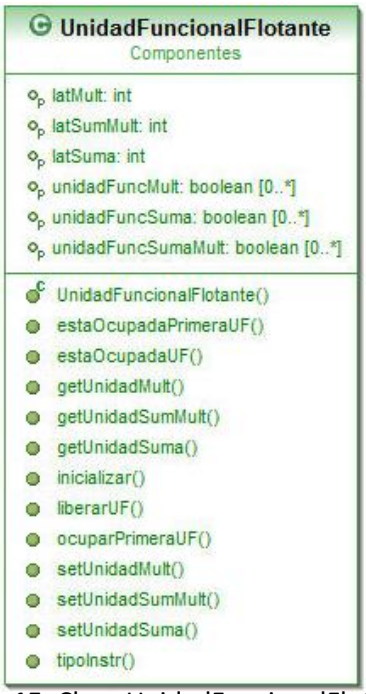


Figura 15: Clase UnidadFuncionalFlotante.

Necesarias para la etapa de ejecución de las instrucciones con aritmética en coma flotante, la unidad funcional presenta un conjunto de vectores con el mismo número de posiciones que el número en el que están segmentadas para poder informar sobre la disponibilidad de las unidades. Para un funcionamiento eficiente, se requiere distinguir si la instrucción que se está tratando es "suma/resta", "multiplicación", "división" o "suma/restaMultimedia" (esta última es unicamente para operaciones flotantes del repertorio AltiVec.
Una vez distinguido el tipo se comprueba el vector correspondiente, teniendo que distinguir entre la primera unidad de la segmentación y el resto. Como era de esperar, la unidad funcional flotante se encarga de actualizar su estado de ocupación cuando una instrucción entra en esa unidad, al igual que debe liberar la unidad cuando una instrucción finaliza su paso por ella.

4.1.6 Unidad de detección y control de riesgos

Decimos que existe un "riesgo" cuando por cualquier impedimento, una instrucción no puede ejecutar su etapa en el ciclo correspondiente.
Dado que se dispone de un simulador segmentado en 5 etapas ya explicadas anteriormente, en el simulador DLX, encontraremos los siguientes tipos de riesgo.

Figura 16: Clase UnidadDeteccionRiesgos.

Riesgos

Debido a la ejecución de instrucción bajo un pipeline nos encontramos con que existen tres tipos de riesgos, estructurales, de control y de datos. En el proyecto tratamos todos ellos con una unidad de detección de riesgos que será constantemente consultada por el planificador.

Riesgos estructurales
Comencemos por los riesgos estructurales, el diseño de circuitería del DLX da lugar en ocasiones a que una instrucción no pueda ejecutar su etapa correspondiente y se bloquee, quedándose en un estado de parada. Este riesgo ocurre cuando 2 instrucciones tratan de acceder a la misma unidad de la segmentación. Pudiendo solo

haber una instrucción en una unidad, la más antigua en el pipeline accederá primero, todas las siguientes deberán esperar a que la primera instrucción libere la unidad. En el simulador se comprueba sencillamente mirando las etapas en las que se encuentran las instrucciones anteriores a la que se está tratando en ese momento. Dicha comprobación se realiza al final del tratamiento de la instrucción permitiéndole o no pasar a la etapa siguiente que le corresponde.
Para controlar este tipo de riesgos a nivel de ejecución es necesario además el uso de una instancia de "UnidadFuncionalFlotante" para saber en la etapa de ejecución que unidades funcionales si están segmentadas están ocupadas y en que subetapas de ejecución.

Riesgos de control
Los riesgos de control tienen lugar principalmente en las instrucciones de salto. El pipeline continúa su ejecución por un camino que no tendría por qué ser el correcto. Para solucionar en parte este riesgo o mejorar el rendimiento del DLX se conoce la dirección del salto y la condición de salto en la etapa decodificación, anticipando incluso hasta la etapa de decodificación un dato en el caso de dependencia. De esta manera solo se pierde una instrucción.
En el caso de que el DLX haya planificado una instrucción y el salto se tome, debe eliminarla en el pipeline. Por ello, anulamos dicha instrucción y lo visualizamos mostrando etapas "X". (Véase Capitulo 6: Casos de Uso)

Riesgos de datos
Los riesgos de datos son tal vez los más interesantes a tratar en el simulador. Aparecen cuando el orden de lectura y escritura no sigue el orden que debería marcar el pipeline. Por ejemplo una instrucción requiere para sus operandos el resultado de otra instrucción que aún no ha acabado de ejecutarse, es decir una carga de datos o una operación aritmética.
La importancia del tratamiento de estos riesgos reside en el hecho de que si no se soluciona el número de ciclos por instrucción aumenta considerable por un exceso de paradas.

- **LDE (lectura después de escritura, del inglés RAW)**

  El primero de estos riesgos es lectura después de escritura, ocurre que en la etapa decodificación se intenta recuperar los datos de los registros fuentes requeridos por la instrucción pero es posible que alguna instrucción anterior tenga como registro destino alguno de estos registros pero aun no haya realizado su etapa de reescritura (WB). Por ello, el dato leído del Banco de Registros, no estaría actualizado, por lo que tomaría un valor antiguo, por lo tanto erróneo. En lugar de introducir paradas, se tratara de anticipar los datos que dependan de instrucciones anteriores, es lo que se denomina cortocircuito ("forwarding" en inglés). En el simulador trataremos de igualar esa función. Primero identificando la dependencia en decodificación de manera que si el dato pudiera anticiparse a la etapa ejecución desde alguna etapa finalizada ejecución anterior o etapa memoria, no se haría parada, si el dato no pudiera

anticiparse daría lugar a un estado ID parada, hasta poder anticipar el dato. Por lo tanto esta comprobación se efectúa en cada ciclo.
La unidad de detección de riesgos deberá por lo tanto comprobar en primer lugar si una instrucción anterior escribe en algún registro fuente de la instrucción que la llama. Para ello y dado que empleamos 3 tipos de registros, entero, flotante y multimedia, debe verificarse si coincide el valor del registro y el tipo del registro, en tal caso y si es posible se realiza la anticipación.
Observación:
No podemos anticipar el dato cuando una instrucción situada justo después de una instrucción tipo Load (instrucciones de carga de memoria a banco de registros), intenta leer el registro en el que el Load escribe. Ya que no lo tendremos accesible hasta que la instrucción load haga su etapa de Memoria. Igualmente existe parada cuando una instrucción flotante depende de una instrucción cuya unidad funcional tiene una latencia mayor de 1, en el caso de que sean consecutivas, mayor de 2 si hay una instrucción entre medias, y así consecutivamente.

- **EDE (escritura después de escritura, del inglés WAW).**

  Riesgo que aparece cuando una instrucción más antigua sobreescribe un dato en un registro destino del banco de registros que ya reescribió en el mismo destino otra instrucción más reciente. En un procesador DLX simple esto no ocurre ya que todas acaban en el mismo orden en el que entraron, pero al añadir las instrucciones en coma flotante, que el simulador DLX posee, puede ocurrir que las instrucciones flotantes sobrescriban con datos incorrectos.
  Véase el siguiente ejemplo, una división flotante tardara 10 ciclos en recorrer el pipeline, la siguiente instrucción, una suma flotante tarda solo 6 ciclos, ambas escriben en el registro destino F10, a partir del ciclo 10 el dato que se recupera del banco de registros será erróneo.
  Para evitar este riesgo en lugar de añadir paradas que volverían menos eficiente el simulador DLX. Decidimos añadir la inhibición de escritura. Para ello cada instancia de "Instrucción" lleva una variable que informa de si se debe o no actualizar el banco de registros con su dato destino. Mediante la unidad de control de riesgos, cuando una instrucción realiza su etapa de reescritura, si escribe en algún registro que alguna instrucción más antigua que no ha finalizado, la instrucción que acaba de actualizar el banco de registros, inhibirá las instrucciones más antiguas con el mismo registro destino.

La unidad de detección de riesgos posee un método que comprueba si deberá o no haber una parada para una instrucción dada, dicho método se encarga de comprobar cualquiera de los riesgos anteriormente explicados.

4.1.7 Estadísticas:

Las estadísticas llevan un contador de todo lo ocurrido en el pipeline, y son actualizadas a lo largo de la ejecución del planificador.

Dentro de las estadísticas que se muestran encontramos todo lo que pueda resultar interesante a la hora del estudio del rendimiento de un simulador DLX y del código cargado. Comenzando por visualizar el número de ciclos que han sido ejecutados hasta el momento y el número de instrucciones, de estos dos elementos se extrae el CPI, ciclos por instrucción.

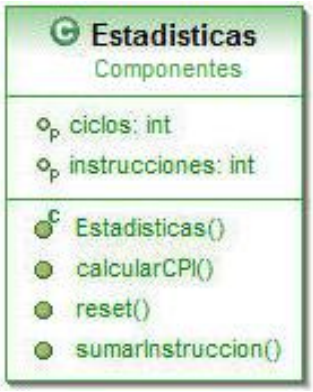


Figura 17: Clase Estadísticas.

A continuación se muestra información sobre el hardware que compone el simulador en momento de ejecución, esto sería la latencia de cada unidad funcional. Y si se encuentra o no activada la anticipación, aunque en nuestro caso, la anticipación no puede desactivarse, esta opción deja abierta la posibilidad a ampliaciones futuras, haciéndolas más sencillas de implementar. Siguiendo los elementos mostrados por las estadísticas se encuentran los riesgos donde se describen la cantidad de paradas por riesgos LDE y EDE, paradas por riesgos estructurales o de control.
Finalmente se lleva la cuenta de los saltos tomados o no tomados, las instrucciones de accesos a memoria de datos y la cantidad de instrucciones flotantes que se han ejecutado en el pipeline. Todos estos datos se recuperan durante el paso por el pipeline quedando justificado porqué las estadísticas se modifican desde el planificador.

4.1.8 Gráficos

4.1.8.1 Gráfico

La pestaña gráfico que encontramos en el panel principal del simulador, nos permite comprobar la ruta de datos actual que está siguiendo el pipeline.
Al inicio simplemente se carga una imagen de un procesador DLX, que se encuentra modificada frente al modelo común, dado que el simulador incorpora unidades funcionales. Y además se requiere modificar la ruta de datos a razón de que las instrucciones multimedia emplean tres registros fuente en la unidad que se acaba de describir. Por ejemplo esto desencadena que se puedan leer tres registros del banco en una misma etapa de decodificación.

Figura 18: Clase Grafico.

El mecanismo del gráfico es sencillo, en cada de la iteración según la etapa que corresponda, se dibuja cada instrucción por separado para su visualización.

- **Etapa IF**
  Se trazan las líneas que corresponden a la etapa, y en particular se escribe el valor del pc sobre el gráfico. La parte importante, o en cierto modo distinta, entre las etapas IF, que se visualizara mediante la imagen de procesador, resulta ser si se ha tomado o no el salto. El multiplexor responsable de nuevo PC muestra la ruta superior indicando que el salto ha tenido lugar.

- **Etapa DEC**

  Durante esta etapa se muestran las entradas al banco de registro y las salidas hasta el buffer ID/EX. Por otra parte si la instrucción que se encuentra en esta etapa se tratase de un salto, se podría observar el cálculo de la dirección destino de dicho salto en lugar de la ruta normal de datos.

- **Etapa EX**

  La etapa EX presenta la ruta de datos para las etapas de ejecución de las instrucciones. Lo interesante de esta etapa y que destacamos a continuación es la visualización de la anticipación o no y las unidades funcionales a nivel internas.

  En cuanto a la anticipación se muestra la distinción de la ruta de datos, como en otras etapas, mediante los multiplexores que seleccionan el dato. La implementación comprueba si hubo o no, anticipación y por parte de que unidad (memoria o ALUs). Se añade un multiplexor extra respecto a un DLX simple a razón de las instrucciones multimedia.

  Como se aprecia en el gráfico disponemos de 3 unidades funcionales distintas, la ruta de datos accede a las unidades. En este punto solo explicaremos la unidad funcional de enteros, la cual solo tarda un ciclo en obtener el resultado. Se muestra por tanto las entradas a la ALU de datos y las salidas al buffer EX/MEM.

  Respecto a las 2 otras unidades funcionales disponen de una clase propia donde las detallamos. A nivel de gráfico visualizamos la entrada a estas unidades sólo en la etapa EX1 (dado que disponen de varias etapas de ejecución) y la salida se visualiza únicamente en la última etapa de ejecución.

- **Etapa MEM**

  A nivel de código se comprueba de que unidad funcional debe tomarse el dato, mostrando la ruta que se toma mediante el multiplexor añadido al DLX sencillo al incorporar instrucciones de coma flotante y multimedia. Esta etapa permite además visualizar si el dato accede o no a memoria o si va directamente al buffer MEM/WB

- **Etapa WB**

  De nuevo mediante el multiplexor de esta etapa se verifica si el dato manejado procede de la memoria o de la ALU. El código comprueba el origen del dato y pinta la ruta correcta en el gráfico.

  Visualiza la ruta de datos que accede al banco de registros para actualizarlo.

Durante todas las etapas, se muestra en la parte superior de cada una de ellas la correspondiente instrucción, si la hubiera.

4.1.8.2 Gráfico Flotante

Esta clase permite mostrar el estado interno de la unidad funcional de coma flotante. En concreto existen 3 unidades flotantes dentro, una para suma/resta, otra para multiplicación y otra para división (no segmentada). En el gráfico el usuario puede observar el estado de éstas, simplemente desplazando el puntero encima de la unidad funcional inferior.

A nivel de código se consulta la latencia de las instrucciones en coma flotante y en función de esa latencia se dibujan tantas fragmentos como latencia-1, cada fragmento representa el buffer donde se van almacenando los resultados obtenidos. El hecho de que haya un buffer de menos se debe a que el ultimo buffer es EX/MEM para cualquier unidad funcional.

Figura 19: Clase GraficoFlotante.

Para el usuario iremos marcando la ruta de datos en función de la etapa de ejecución que se encuentre, y se visualizaran las instrucciones que se están llevando a cabo dentro de las unidades funcionales en orden de ejecución.

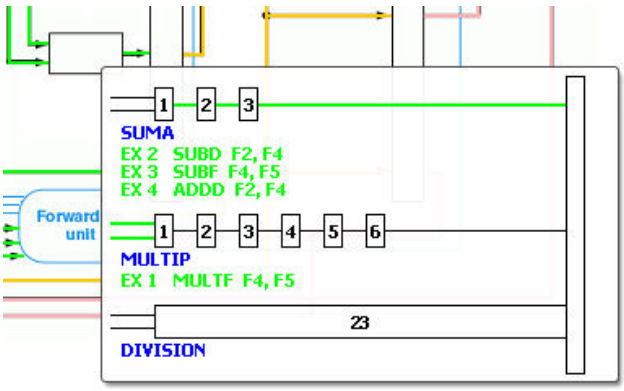


Figura 20: Ejemplo del estado de las Unidades Funcionales Flotantes, durante la ejecución de varias instrucciones.

4.1.8.3 Gráfico Multimedia

Esta clase es completamente análoga a la vista en graficoFlotante, solo que simplemente visualiza una única unidad flotante para la Suma/resta con multiplicación. En el gráfico el usuario puede observar el estado de ésta, simplemente desplazando el puntero del ratón por encima de la unidad funcional inferior.

Figura 21: Clase GraficoMultimedia.

Al igual que la unidad flotante, se consulta los ciclos de latencia de la unidad funcional Suma+Multiplicacion, y en función de esa latencia se dibujan tantas fragmentos como latencia-1, cada fragmento representa el buffer donde se van almacenando los resultados obtenidos. El hecho de que haya un buffer de menos se debe a que el ultimo buffer es EX/MEM para cualquier unidad funcional.

Igual que ocurre en el gráfico flotante, se visualiza cada etapa y su instrucción correspondiente de la ejecución, en el orden descendente.

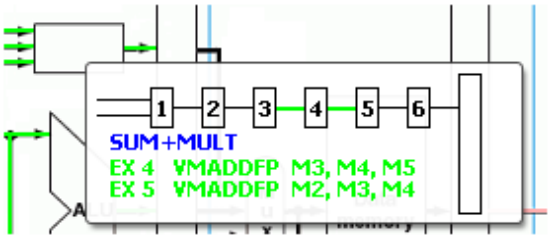


Figura 22: Ejemplo del estado de la Unidad Funcional Suma+Multiplicación, durante la ejecución de varias instrucciones.

4.2 Módulo Parser:

Índice:

4.2.1 Compilador

El compilador es el encargado del reconocimiento de las directivas de un DLX.

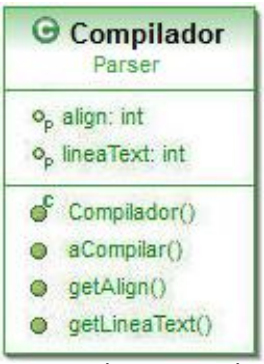


Figura 23: Clase Compilador.

La técnica que utilizamos básicamente, es leer línea a línea el código introducido por el usuario y cada una de esas líneas, será introducida en el "Parser".
Al comienzo de la compilación, el compilador se encargará de leer las directivas del programa. Las primeras directivas que se deben leer son "text" o "data", sino el compilador lanzara un error.
Si la primera directiva que lee es ".text", esto determinará el comienzo de la región de código y el Parser pasara a reconocer las instrucciones. En caso contrario, si lee la directiva ".data" determinará el comienzo de la región de datos y pasará a un estado, donde ira reconociendo las directivas del DLX.
El proceso de reconocimiento de las directivas, dentro de la región de datos es similar para todas ellas, y consiste en dividir la línea en "tokens" y reconocer el primer token como la palabra clave que identifica a la directiva, las cuales van siempre precedidas de un punto, y en función de la directiva enviar la línea fragmentada en tokens a un subparser.

El subparser, en función de la directiva que le llegue, se encarga de comprobar los tipos, el número de argumentos, y devolver la información correcta al compilador para poder introducir los datos en memoria principal.

Directivas

Dentro de la región de datos nos podemos encontrar con nueve posibles directivas (las directivas están detalladas en el anexo final):

- **"align n":** Alinea los datos que posteriormente se van a introducir, de forma que sean cargados en la parte más alta de la dirección, con los n bits más bajos puestos a cero.

- **"ascii":** Reconoce una cadena de caracteres encerrada entre comillas y los guarda en memoria.

- **"asciiz":** Reconoce una cadena de caracteres encerrada entre comillas y los guarda en memoria. Al final de la cadena guarda el byte "NULL".

- **"byte":** Reconoce una secuencia de bytes que se han declarado y los guarda en memoria consecutivamente.

- **"double":** Reconoce una secuencia de números "doubles" que se han declarado y los guarda en memoria de forma consecutiva.

- **"float":** Reconoce una secuencia de números expresados en coma flotante, que se han declarado y los guarda en memoria de forma consecutiva.

- **"word":** Reconoce una secuencia de palabras que se han declarado y los guarda en memoria consecutivamente.

- **"space n":** Reserva n posiciones de memoria avanzando el puntero que apunta a que dirección de memoria guardar datos.

- **"global etiqueta":** Permite que una etiqueta pueda ser referenciada en cualquier parte de la región de código, estas etiquetas son usadas en la región de código para poder determinar el lugar donde queremos hacer saltos en la ejecución.

Dentro de la región de datos, también podemos hacer una declaración de etiquetas para guardar una posición de memoria, y su posterior uso en la región de código. Estas etiquetas van sucedidas de dos puntos y podrá tomar el nombre que desee el usuario. El funcionamiento de estas etiquetas es el siguiente, el Parser detectará que la línea de código finaliza por dos puntos y añadirá el nombre de la etiqueta a una tabla donde se guardará también la posición de memoria del siguiente dato que este declarado.

Instrucciones
Además, el compilador también es el encargado de reconocer las instrucciones de un DLX y ALTIVEC:

Una vez identificada la declaración de sentencia mediante .text el Parser entra en un estado de reconocimiento de las instrucciones que forman el repertorio DLX y el repertorio multimedia ALTIVEC. En este estado el Parser recuperara línea a línea el código introducido por el usuario y cada línea será fragmentada en el número de tokens que forman la instrucción siendo el primer token el código de operación, y los siguientes el operando destino y operandos fuente.
El Parser comprueba en primer lugar si el código de operación leído corresponde con alguno del repertorio DLX o AltiVec. En este punto destacamos que las instrucciones y su compilación están distribuidas en función del repertorio al que pertenezca y del tipo que sean, véase la explicación siguiente (las instrucciones están detalladas en los apéndices I: Repertorio de instrucciones DLX, y en el apéndice III: Repertorio de instrucciones AltiVev)

En cuanto al repertorio DLX las instrucciones están agrupadas según su funcionalidad. Disponemos de siete conjuntos disjuntos:

- **"dataTransfersCommands":** Contiene todas las instrucciones que transfieren datos de un registro del banco de registros a otro o de un registro a memoria o de memoria a un registro. Las resumimos como instrucciones de carga ("lb", "lbu", "lh", "lhu", "lw", "lf", "ld"), guardado ("sb", "sh", "sw", "sf", "sd") y transferencia ("movi2s", "movs2i", "movf", "movd", "movfp2i", "movi2fp").

- **"aritmethicLogicCommands":** Compuesto por todas las instrucciones aritméticas con signo ("add", "sub", "mult", "div"), las instrucciones aritméticas sin signo ("addu", "subu", "multu", "divu"), las instrucciones lógicas ("and", "or", "xor") y las instrucciones de comparación sobre registros ("sll", "srl", "sra", "slt", "sgt", "sle", "sge", "seq", "sne").

  Todas ellas operan con dos registros fuente y almacenan en el banco de registros sobre el registro destino reconocido.

- **"arithmeticLogicInmediateCommands":** Formado por todas las instrucciones aritméticas con signo ("addi", "subi"), las instrucciones aritméticas sin signo ("addui", "subui", "multui", "divui"), las instrucciones lógicas ("andi", "ori", "xori") y las instrucciones de comparación sobre registros ("slli", "srli", "srai", "slti", "sgti", "slei", "sgei", "seqi", "snei"). Las cuáles operan con un registro fuente y un inmediato, cuyo rango queda especificado en el anexo final, y almacenan en el banco de registros sobre el registro destino reconocido.

- **“controlCommands”:** Es el conjunto de instrucciones que controlan el valor de PC para realizar saltos y retornos (“beqz”, “bnez”, “bfpt”, “bfpf”, “j”, “jr”, “jal”, “jalr”, “trap”, “rfe”) Y de las instrucciones que no realizan nada o carga un inmediato directamente al banco de registros(“nop”, “lhi”).

- **“fpCommands”:** Representa las instrucciones aritméticas que llevan por sus dos operandos fuente y su destino elementos en coma flotante, con simple precisión (“addf”, “subf”, “multf”, “divf”) o doble precisión (“addd”, “subd”, “multd”, “divd”).

- **“convertFPCommands”:** Son aquellas instrucciones que permiten realizar conversiones de los elementos contenidos en el banco de registros en todo sentido posible, de un doble a un flotante (“cvtd2f”), de un flotante a un doble (“cvtf2d”), de un doble a un entero (“cvtd2i”), de un entero a un doble (“cvti2d”), de un flotante a un entero (“cvtf2i”) y de un entero a un flotante (“cvti2f”).

- **“setOnComparisonFPCommands”:** Compuesto por todas las instrucciones que modifican el registro especial de los elementos en coma flotante “FPSR” en función de la comparación (mayor, menor, igual) que se realiza entre los dos operandos fuente de tipo coma flotante con simple precisión (“ltf”, “gtf”, “lef”, “gef”, “eqf”, “nef”) o doble precisión (“ltd”, “gtd”, “led”, “ged”, “eqd”, “ned”).

En cuanto al repertorio AltiVec, las instrucciones se dispondrán en conjuntos que comparten el mismo formato de instrucción. Disponemos de ocho conjuntos, es decir ocho formatos diferentes:

- **“Multimedia3OperandsWithDestCommands”:** Formado por las instrucciones que requieren de tres operandos, dos registros fuente y un registro destino, todos ellos de tipo multimedia. Entre las instrucciones que lo componen encontramos instrucciones aritméticas, lógicas con signo, sin signo, modulares, saturadas, de comparación, en distintos tamaños para los operandos (byte (8 bits), halfword (16 bits), Word (32 bits)), y cuyos elementos pueden ser enteros o flotantes. (Para más detalle consultar el anexo final de AltiVec) (“vaddubm”, “vadduhm” ,”vadduwm”, “vaddhum”, “vaddhus”, “vaddhss”, “vaddfp”, “vaddcuw”, “vaddubs”, “vaddsbs”, “vadduhs”, “vaddshs”, “vadduws”, “vaddsws”, “vand”, “vandc”, “vavgsb”, “vavguh”, “vavgsh”, “vavguw”, “vavgsw”, “vcmpbfp”, “vcmpequb”, “vcmpequh”, “vcmpequw”, “vcmpeqfp”, “vcmpgefp”, “vcmpgtub”, “vcmpgtsb”, “vcmpgtuh”, “vcmpgtsh”, “vcmpgtuw”,

"vcmpgtsw", "vcmpgtfp", "vcmpgefp", "vcmpgtub", "vcmpgtsb", "vcmpgtuh", "vcmpgtsh", "vcmpgtuw", "vcmpgtsw", "vcmpgtfp", "vmaxub", "vmaxsb", "vmaxuh", "vmaxsh", "vmaxuw", "vmaxsw", "vmaxfp", "vmrghb", "vmrghh", "vmrghw", "vmrglb", "vmrglh", "vmrglw", "vminub", "vminsb", "vminuh", "vminsh", "vminuw", "vminsw", "vminfp", "vmuleub", vmulesb", "vmuleuh", "vmulesh", "vmuloub", "vmulosb", "vmulouh", "vmulosh", "vnor", "vor", "vpkuhum", "vpkuwum", "vpkpx", "vpkuhus", "vpkshss", "vpkuwus", "vpkswss", "vpkuhus", "vpkshus", "vpkuwus", "vpkswus", "vrlb", "vrlh", "vrlw", "vslb", "vslh", "vslw", "vsl", "vslo", "vsrb", "vsrh", "vsrw", "vsrab", "vsrah", "vsraw", "vsr", "vsro", "vsububm", "vsubuhm", "vsubuwm", "vsubfp", "vsubcuw", "vsububs", "vsubsbs", "vsubuhs", "vsubshs", "vsubuws", "vsubsws", "vsum4ubs", "vsum4sbs", "vsum4shs", "vsum2sws", "vsumsws", "vxor".

- **"Multimedia4OperandsWithDestCommands":** Contiene las instrucciones que necesitan cuatro operandos, tres registros fuente y un registro destino. Son aritméticas en distintos tamaños para los operandos (byte (8 bits), halfword (16 bits), Word (32 bits)), saturadas o modulares con signo y sin signo y cuyos elementos son enteros o flotantes ("vmaddfp", "vmhaddshs", "vmladduhm", "vmhraddshs", "vmsumubm", "vmsummbm", "vmsumuhm", "vmsumshm", "vmsumuhs", "vmsumshs", "vnmsubfp", "vperm", "vsel").

- **"Multimedia2OperandsWithDestCommands":** Compuesto por las instrucciones que emplean dos operandos, un registro fuente y un registro destino. Estas instrucciones son operaciones monarias sobre el operando fuente, como por ejemplo redondeos ("vrfip", "vexptefp", "vrfim", "vlogefp", "vrefp", "vrfin", "vrsqrtefp", "vrfiz", "vupkhsb", "vupkhpx", "vupkhsh", "vupklsb", "vupklpx", "vupklsh").

- **"MultimediaVsldoi":** Es el conjunto de la instrucción "vsldoi" que utiliza cuatro operandos, dos registros fuente, un inmediato fuente de 4 bits sin signo y un registro destino.

- **"Multimedia3OperandsWithDestAndImnCommands":** Representa a las instrucciones de tres operandos, donde uno es un registro fuente, otro es un inmediato fuente de 5 bits sin signo y otro es el registro destino ("vspltb", "vsplth", "vspltw", "vcfux", "vcfsx", "vctsxs", "vctuxs").

- **“Multimedia2OperandsWithDestAndImnCommands”:** Son las instrucciones que usan dos operandos, un inmediato fuente de 5 bits con signo y un registro destino (“vspltisb”, “vspltish”, “vspltisw”, “vspltisb”, “vspltish”, “vspltisw”).

- **“MultimediaMfvscrMtvscr”:** Como su nombre indica contiene las instrucciones “mfvscr” y “mtvscr” ambas operan con el registro especial “VCSR”, y necesitan un registro destino y fuente respectivamente.
- **“MultimediaLoadsStores”:** Contiene las instrucciones multimedia para cargado y guardado de los registros multimedias, como operandos necesitan, dos registros fuente un registro de tipo R fuente y un registro fuente multimedia, y un registro destino multimedia (“lvebx”, “lvehx”, “lvewx”, “lvxl”, “lvsl”, “lvsr”, “lvx”, “stvx”,”stvebx”,”stvehx”, “stvewx”,”stvxl”).

En el parser encontramos un subparser para cada uno de los conjuntos descritos anteriormente. El hecho de disponer de los subparser simplifica la depuración y mejora el rendimiento, al no comprobar todas las posibles combinaciones de los formatos de todas las instrucciones. Además supone una ventaja a la hora de incorporar si se desease más instrucciones a los repertorios ya parseados.
Cada subparse será el encargado de compilar la información de la instrucción bajo un formato aceptado por el programa principal. Para llamar al subparser correcto el parser identifica el código de operación y le manda al subparser del conjunto al que pertenece el código de operación la línea de código que contendrá la información a analizar.

En el subparser de deberán comprobar el resto de elementos de la línea de código. Diferenciamos dos tipos de subparser. Aquellos que aún tienen que comprobar el código de operación para saber identificar los datos entrantes, y aquellos que directamente comprueban los datos. El primer grupo de datos corresponde exclusivamente al repertorio DLX, dado que para añadir el repertorio AltiVec se optó por separar las instrucciones en función de su formato en lugar de por su tipo o comportamiento para mejorar su eficiencia y la simplicidad del código.

Respecto al primer grupo explicaremos un ejemplo con detalle:
Sea el caso de subparser para el conjunto “fpCommands”, no resulta especialmente complejo implementarlo. Lo primero se comprobaría que el número de elementos de la línea de código es cuatro, código de instrucción, un registro destino y dos registros fuente. Sin embargo cuando se comprueban los datos que forman la instrucción, en este caso son registros flotantes (f0-f31). Deberá tenerse en cuenta el código de instrucción dado ya que si termina en “d” deberá verificarse que el registros f‘X’, ‘X’ sea un entero par y en cualquier caso termine o no por “d” se verificara que el registro

de cada elemento de la instrucción pertenezca al rango [0-31] además de comprobar que los registros sean de tipo F. [39]

Veamos ahora un ejemplo del segundo tipo de subparser, el que no vuelve a chequear la instrucción:
Sea el caso de subparser para el conjunto "aritmethicLogicCommands", también pertenece al DLX aunque éste solo comprobara los registros dado que todas las instrucciones que forman el conjunto poseen el mismo formato y las mismas restricciones, solo se comprobara que el número de elementos que forman la línea de código son cuatro, código de instrucción, un registro destino y dos registros fuente. Y que cada registro sea de tipo R y pertenezca al rango [0-31].

Los dos ejemplos descritos pertenecen al repertorio DLX, al comprobar la mejora que suponía el segundo ejemplo se siguió ese sistema para el repertorio añadido multimedia. De esta manera todo subparser de los conjuntos del repertorio AltiVec solo debe comprobar los registros o inmediatos que forman la instrucción (rangos y tipos).

Detección de errores

El parser también se encarga de detectar errores en el código recuperado. Se reconocen pues varios tipos de errores:

- El caso de que el código de operación no exista, no entran en acción los subparsers y se devuelve la línea del error y la línea de código con el mensaje de que no existe dicha código de operación.

- Si una instrucción tiene más elementos o menos elementos de los necesarios se informará de ello al usuario mediante la línea donde se encuentra el error y la línea de código y el mensaje correspondiente, número de argumentos incorrecto.

- Si el registro reconocido no es el del tipo esperado o no lo acompaña un entero, se informara de la línea del error, de la línea de código y de que es un nombre inválido para el registro.

- Si el entero del registro esta fuera de rango o no reconoce el registro correctamente, se informara de la línea del error, de la línea de código y de que el registro no existe.

- Si el entero del registro no es par y debería serlo, se informara de la línea del error, de la línea de código y de que el registro debe ser par.

- Si se espera un inmediato y éste está fuera del rango esperado, se informara de la línea del error, de la línea de código y de que el rango del inmediato no es correcto.

El parser se encargará de recoger todos los errores sin interrumpir la ejecución y cuando haya analizado todo el campo de código, mostrara al usuario el listado de errores recuperados mediante una nueva ventana permitiéndole corregir los errores resultantes de la compilación.

En resumen, en lo que respecta al parser para las instrucciones de los repertorios DLX y AltiVec, se identifica el código de instrucción leído en la línea, si existiera en alguno de los conjuntos de instrucciones se ordenaría al subparser correspondiente que analizase la línea de instrucción campo por campo, código de instrucción si fuera necesario, registros comprobando su tipo(R,F o M) y rango([0-31]), los inmediatos comprobando su rango según la instrucción, si fueran etiquetas comprobar su existencia y valor correcto. Y en caso de reconocer algún dato incorrecto se detectaría el error, se identificaría y se mostraría al usuario del programa una vez recorrido todo el código.

4.2.2 Información del Dato

Ante la necesidad de devolver la información leída por el Parser referente a cada una de las etiquetas al compilador, surge la clase DatInfo. Esta clase sirve para devolver la información de cada línea que contenga una etiqueta.

Figura 24: Clase DatInfo.

Para ello dispone de dos campos, uno para guardar el nombre de la etiqueta que se ha leído y otro para guardar la información asociada a esa etiqueta como valores de variables o posiciones de memoria.

4.2.3 Información de la Instrucción

Ante la necesidad de devolver al compilador la información leída por el Parser referente a cada una de las instrucciones surge la clase "InstInfo". Esta clase sirve para devolver la información de cada línea que contenga una instrucción.

Figura 25: Clase InstInfo.

Para ello dispone de varios campos, uno para guardar el nombre de la instrucción, otros campos para guardar los diferentes registros fuente y destino que empleará cada instrucción así como inmediato o dirección de salto.

4.2.4 Etiquetas Pendientes

Puede surgir el problema de que se haya declarado una etiqueta como "global" y que aparezca el salto a esa etiqueta antes que la etiqueta en cuestión.

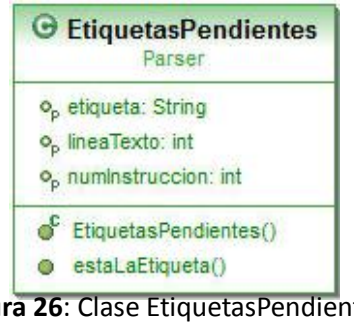


Figura 26: Clase EtiquetasPendientes.

Para ello está la clase “EtiquetasPendientes”, esta clase guarda la Instrucción que tenemos que sustituir y el número de instrucción que tiene asociada la instrucción para cuando aparezca la etiqueta poder actualizar la Instrucción con la dirección de salto.

4.2.5 Errores

Ante la posibilidad de introducir varios idiomas en el simulador y para mantener un orden y claridad ante los errores que pueden surgir al usuario al escribir un programa y compilarlo esta la clase “Errores”. Esta clase nos permite identificar los errores y organizarlos según el tipo de error que sea, también dependiendo del idioma que este seleccionado el error aparecerá en el idioma correspondiente.

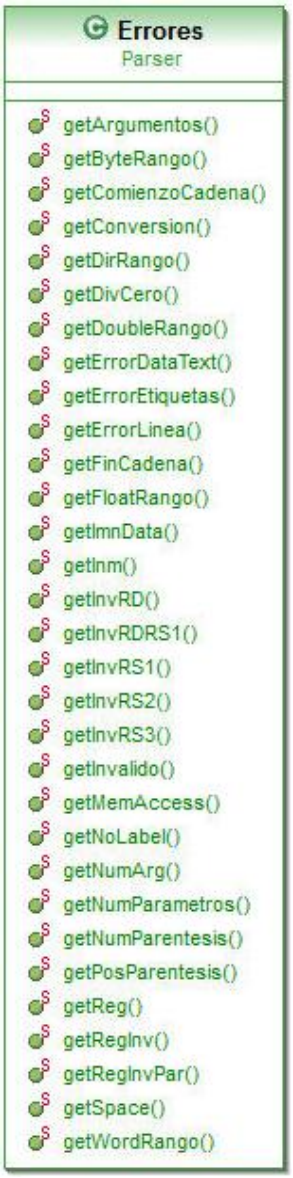


Figura 27: Clase Errores.

A continuación se detallaran los errores que pueden surgir tanto durante la compilación como la ejecución:

ASOCIADOS A DIRECTIVAS:

- **"ErrorDataText":** Aal comenzar a escribir un código la ausencia de alguna de las directivas "data" o "text".

- **"ErrorEtiquetas":** Cuando aparece repetida alguna etiqueta del programa ya sea tanto en la región de código como en la región de datos.

- **"ComienzoCadena":** Al declarar una cadena de texto con las directivas "ascii" o "asciiz" hay algún error al comenzar alguna cadena.

- **"FinCadena":** Al declarar una cadena de texto con las directivas "ascii" o "asciiz" hay algún error al finalizar alguna cadena.

- **"Argumentos":** Error cuando falta algún argumento en alguna directiva.

- **"Conversion":** Cuando al introducir una directiva que introduzca datos como "byte" o "word" y no se pueda hacer la conversión al tipo que corresponda.

- **"DirRango":** Cuando se sale de rango un acceso a memoria.

- **"Space":** Cuando con la directiva "space" se quiere reservar mas direcciones de memoria de las que realmente hay.

- **"ByteRango":** Cuando en la directiva "byte" se introduce un valor que no se puede instanciar a un byte.

- **"FloatRango":** Cuando en la directiva "float" se introduce un valor que no se puede instanciar a un float.

- **"WordRango":** Cuando en la directiva "word" se introduce un valor que no se puede instanciar a un word.

- **"DoubleRango":** Cuando en la directiva "double" se introduce un valor que no se puede instanciar a un double.

ASOCIADOS A INSTRUCCIONES:

- **“Invalido”:** Cuando se introduce una instrucción que no corresponde al repertorio DLX-AltiVec.
- **“Reg”:** Cuando se halla un error porque no se encuentra el registro introducido.
- **“RegInv”:** Cuando se introduce un registro que no es válido.
- **“RegInvPar”:** Cuando se introduce un registro que no es válido porque no es par.
- **“Inm”:** Cuando se introduce un valor que no es un inmediato.
- **“NumArg”:** Cuando hay un error con el número de argumentos asociados a una instrucción.
- **“InvRD”:** Cuando se introduce un nombre invalido para el registro destino.
- **“InvRS1”:** Cuando se introduce un nombre invalido para el registro fuente uno.
- **“InvRS2”:** Cuando se introduce un nombre invalido para el registro fuente dos.
- **“InvRS3”:** Cuando se introduce un nombre invalido para el registro fuente tres.
- **“InvRDRS1”:** Cuando se introduce un nombre invalido para el registro destino o fuente uno.
- **“NumParentesis”:** Cuando se introduce un numero incorrecto de paréntesis.
- **“PosParentesis”:** Cuando hay paréntesis en posiciones incorrectas.
- **“ImnData”:** Cuando el offset o inmediato introducido es incorrecto.
- **“NoLabel”:** Cuando se usa una etiqueta que no ha sido declarada previamente.

ASOCIADOS A LA EJECUCION:

- **“MemAccess”:** Cuando se produce un acceso a una dirección de memoria invalida durante la ejecución.

- **“DivCero”:** Cuando se produce una división por cero durante la ejecución.

4.3 Módulo Instrucciones:

Índice:

4.3.1 Conversiones

La clase conversiones es la responsable de la transformación de los datos entre distintos formatos para manipulación de la información dentro del simulador DLX de la manera más sencilla posible.

Figura 28: Clase Conversiones

El por qué de esta clase rige en el hecho de que debemos emplear un lenguaje de alto nivel como es java respecto a un lenguaje de bajo nivel que es el repertorio DLX y AltiVec. El funcionamiento del DLX requiere el uso de datos en forma de bits, sin embargo y por lo general para facilitar las operaciones en el simulador de java, empleamos los tipos primitivos int, float y string de java a nivel de código.

Esta clase estática contiene las principales y necesarias conversiones.

Conversiones entre enteros y binario

Entre ellas se encuentra, la opción de transformar un entero ('int') al dato binario en 32 bits, y de binario de 32 bits a entero. La ventaja a la hora de emplear java, es que los datos int en la máquina de java están representados en 32 bits en complemento a 2, con ello se logra que ambos rangos de números coincidan y que las transformaciones a binario sean directas. El elemento binario que devuelven las conversiones o al que entran, es en realidad una cadena de caracteres conteniendo solo '0' y '1'.

En relación a estas conversiones de enteros destacamos también la necesidad de emplear enteros grandes "long" en operaciones en las cuales se necesita saber si hubo desbordamiento y por lo tanto se incrementa el rango mediante enteros long, por lo que se incluyen también en esta clase conversiones de long a binario y viceversa.

Dado que el repertorio DLX y AltiVec cuentan ambos con operaciones sin signo, es necesario la existencia de los métodos anteriores pero esta vez para tratar los datos sin signo, tanto métodos sin signo int a binario como long a binario y viceversa.

Conversiones entre coma flotante y binario

Prosiguiendo con la enumeración de transformaciones necesarias, debemos transformar igualmente los flotantes de simple y doble precisión a binario y viceversa, para ello seguimos el estándar IEEE 754 de representación de números en coma flotante. A ambos métodos les entra un float o un double según la precisión requerida, devolviendo cadenas de caracteres de 32 bits en simple precisión o 64 bits si es doble precisión. Al igual que ocurría con los enteros coincide que la representación en java de los flotantes de simple precisión es de 32 bits y de los flotantes de doble precisión es de 64 bits y estos métodos son directos en cuanto a obtener el dato binario.

IEEE 754 descompone la cadena de bits, en signo, exponente y mantisa. Incluyendo los números denormalizados. Son necesarios métodos auxiliares que analicen el exponente y la mantisa en función de la precisión requerida.

Los métodos de conversión binaria a flotante requerían pues saber el número de bits que componían la mantisa y exponente, en función de si se trata de simple o de doble

precisión en concreto, para simple precisión el exponente son 8 bits y la mantisa 23 y en doble precisión el exponente de 11 bits y la mantisa de 52 bits.
En relación con el tratamiento de los elementos flotantes en doble precisión, al representarse con una cadena de 64 bits, ha de fragmentarse para guardarse en dos registros contiguos de 32 bits simulados por el proyecto, por ello en esta clase se añade un método para obtener la primera parte de un doble o bien la segunda.

Para finalizar con la conversión de datos, se requiere un método que devuelva un byte como la cadena binaria que lo representa, dicha operación solo se referencia en el parser a la hora de añadir datos a la memoria en forma de bytes o caracteres.

Conversiones entre multimedia
Al añadir las instrucciones multimedia al simulador fue necesario incorporar varios métodos nuevos, en primer lugar un método que devolviese un flotante en forma de cadena de 32 bits truncado a 0 dado que algunas instrucciones indican que debe cumplirse esa condición. A continuación y para mejorar la comodidad de comprensión del código, se generan tres nuevos métodos que fragmentan el contenido de los registros multimedia de 128 bits en partes tamaño de byte 8 bits * 16 , halfword 16 bits * 8 y Word 32 bits *4.

Extensión de signo
Para acabar con este conjunto, una operación necesaria y presente en la arquitectura DLX es la extensión de signo, este método permite extender el signo de cualquier valor hasta 32 bits.

Conversiones para visualización
Todas las operaciones y transformaciones vistas hasta ahora eran necesarias para la ejecución de las instrucciones y el manejo de los datos en éstas o de conversiones de los datos recuperados al compilar el código del parser. A partir de ahora entramos en un nuevo conjunto de métodos que comentaremos por encima dado que hacen referencia a la parte de la interfaz gráfica del proyecto.
A nivel de usuario a la hora de observar los datos contenidos en la memoria y en el banco de registros tanto de enteros como multimedia como flotantes, son necesarias las conversiones entre los tipos representantes y los que se desean visualizar. Esto da lugar a que los tres tipos que representan los datos, int float/double y strings que ya se mencionaron en cada apartado (memoria y banco de registros) requieran diferentes métodos de transformación para su visualización, por ello se añaden funciones que los transformen en datos hexadecimales, facilitando así la comprensión de los datos.

4.3.2 Tipos de instrucción (Introducción)

Las instrucciones se encuentran codificadas en las instancias de "Instrucción" sin embargo, para el pipeline y a la hora de representar el comportamiento del procesador DLX se toma la decisión de emplear diferentes clases para llevar a cabo las diferentes etapas por las que pasa la instrucción en el pipeline. Para mantener una cierta estructura lógica y no hacer un código pesado, el repertorio DLX y AltiVec de instrucciones simuladas, ha sido fragmentado por tipos.
Comenzando por los tipos DLX nos encontramos ante 3 tipos diferentes, así establecidos por la arquitectura DLX, hablamos por supuesto del tipo R, tipo I y tipo J.
A continuación, encontramos los conjuntos que son de tipo multimedia y por lo tanto salvo que se indique alguna peculiaridad emplearán solo registros multimedia indicados como tipo M en el simulador.

Subráyese el hecho de que las instrucciones que forman los conjuntos multimedia han sido distribuidas de manera arbitraria por los componentes del equipo de proyecto, no están indicadas en dichos conjuntos bajo ningún convenio, sino que se decidió de esta manera para llevar a cabo una implementación eficiente, sencilla y comprensible de las instrucciones.

4.3.3 Tipo R

El conjunto de instrucción R, representa todas aquellas instrucciones del repertorio DLX cuyas operaciones se efectúan exclusivamente sobre registros, ya sean enteros o flotantes o los registros especiales el entero de interrupción IAR o el flotante FPSR. Entre todas estas instrucciones se distinguen diferentes conjuntos basados en su comportamiento.

Aritmetico/lógicas

Un primer conjunto formado por las instrucciones aritméticas o lógicas tanto en flotante como en enteros.
Este conjunto dispone de 2 registros fuente y un registro destino, los 3 siempre del mismo tipo. Empleando este mismo formato de operandos encontramos las operaciones de desplazamiento a izquierda y derecha. El comportamiento de estas operaciones es similar al visto en el conjunto anterior, se opera sobre dos datos fuente y el resultado se almacena en el registro destino.

Comparaciones

Muy similar a estos dos conjuntos, existe un grupo de instrucciones comparativas que emplean los mismos operandos, en función de si es mayor o menor o igual o cualquier combinación entre éstas, se almacena un 1 o un 0 en el registro destino.
En cuanto a las comparaciones en punto flotante el conjunto de instrucciones que las compara realiza las mismas comparaciones respecto a los datos de los registros flotantes en doble y simple precisión sin embargo el destino es esta vez el registro especial de los flotantes (FPSR) donde según la comparación se guarda un 1 o un 0.

Figura 29: Clase TipoR

Conversiones y movimientos

Finalmente existen dos conjuntos más dentro de las instrucciones tipo R compuestos por las instrucciones con un registro destino y un registro fuente. Estos dos conjuntos permiten la conversión de datos entre flotantes (simple y doble precisión) y enteros o directamente mover el dato contenido en un registro a un registro destino pudiendo ser flotante o entero. (Estos conjuntos no emplean registros multimedia).

Comportamiento de las Tipo R en el Pipeline

La explicación del comportamiento genérico de las instrucciones tipo R del repertorio DLX permite justificar el trato que describiremos a continuación de los distintos conjuntos que contienen las instrucciones en el momento de ejecutar cada etapa por la que pasa una instrucción. Por ello se dispone de una clase "TipoR" responsable de evaluar la operación y entonces llevar a cabo las correspondientes acciones. Para ello la clase dispone de un método distinto por cada etapa no de parada que ejecuta una instrucción en su paso del pipeline.

Etapas

- **Etapa IF**
  Comenzando por la etapa IF, etapa de identificación de la instrucción, en el caso de las instrucciones tipo R, y veremos que en realidad esta etapa es igual para todos los tipos que veremos a continuación, se identifica el tipo de operación a realizar y se modifican los valores de la instrucción para llevar la información necesaria al planificador, como por ejemplo si se trata o no de una operación de doble precisión o el número de ejecuciones que forman su etapa de ejecución en función de la latencia de la unidad funcional sobre la que opere.
  En casos particulares, es decir cuando el destino es un registro especial , es el caso de las comparaciones en coma flotante se aprovecha para indicar como destino en la operación el registro flotante que a la hora de compilar no existía dado que era implícito a la instrucción. Podría haberse añadido en el compilador pero esto implicaba que el parser realizará un análisis semántico lo que se consideró innecesario dado que simplemente se requería reconocer correctamente las instrucciones.

- **Etapa ID**
  Prosiguiendo con el orden en que las etapas tienen lugar en el pipeline, durante la etapa decodificación ID, tienen lugar la recuperación de los datos de los registros e identificación de los registros accedidos para la visualización en la

tabla del pipeline. Independientemente de si luego se anticipa o no el operando, los datos son recuperados del banco de registros del tipo de registros adecuado.

- **Etapa EX**
  Avanzando a etapa de ejecución, en el caso de las instrucciones tipo R es la parte más interesante, dado que todas las instrucciones de este tipo implican el uso de la ALU. Llegado a esta etapa es el punto donde se emplean las conversiones (ver conversiones) para ejecutar todas las operaciones de la ALU. Salvo las instrucciones del conjunto responsable de trasladar el contenido de un registro a otro tipo de registro, todas las instrucciones tipo R emplean su unidad funcional correspondiente para obtener el resultado que quedara almacenado en la instancia de "Instrucción". Como recordatorio, esta parte de la instrucción solo tendrá lugar en el simulador cuando la última etapa de ejecución de la unidad funcional se ejecute.

- **Etapa MEM**
  En cuanto a la etapa de acceso a memoria, las instrucciones tipo R no acceden a memoria ni realizan nada en esta etapa simplemente pasan por ella.

- **Etapa W**
  Para finalizar con las instrucciones tipo R, en la etapa de reescritura se devuelve únicamente el resultado que se obtuvo en la etapa de la ejecución en función del tipo de operación que sea, entero o flotante en simple o doble precisión.

4.3.4 Tipo I

Las instrucciones tipo I tienen ese nombre por el hecho de que todas ellas emplean un elemento inmediato. A razón de esto se ha decidido clasificar estas instrucciones en un solo grupo sin embargo el comportamiento individual de cada una no se puede generalizar.

Se observan los mismos subconjuntos de las instrucciones tipo R pero esta vez operando con un inmediato fuente, en lugar de con un dato obtenido de un registro fuente. Por ello se dispone de los siguientes conjuntos.

Aritmético/lógicas

En primer lugar el conjunto de operaciones aritméticas y lógicas que emplean un registro destino para el resultado de la operación correspondientes y 2 operandos fuente, uno obtenido de un registro del banco de registros y el otro introducido como inmediato a la hora de compilar el código.

Comparaciones

En segundo lugar las operaciones de comparación pero empleando como segunda fuente el inmediato.

Desplazamientos

En tercer lugar el conjunto de operaciones que realizan un desplazamiento en función del valor del inmediato. En este punto, cabe destacar que no se emplean registros fuente ni destino de tipo flotante, es decir solo se emplean inmediatos sobre operaciones enteras.

Accesos a Memoria

Con respecto a las instrucciones tipo R, las tipo I llevan un conjunto de instrucciones de acceso a memoria, que emplean el inmediato junto con el contenido de un registro fuente para calcular la dirección de memoria a la que se accede, ya sea para cargar o almacenar datos.

Saltos

Por último existen operaciones de salto cuyo inmediato es en verdad en el simulador la etiqueta que el usuario ha introducido codificada a la posición de la instrucción a la que debe saltar. Estas instrucciones realizan una comparación y si se cumple la condición marcada se salta al inmediato. Existen 2 excepciones en este paquete de instrucciones, JR y JALR, las cuales no emplean ningún inmediato pero se clasifican como instrucciones tipo I.

Figura 30: Clase TipoI.

Comportamiento Tipo I en el Pipeline

El comportamiento que sigue este tipo en cada etapa del pipeline.

Etapas

- **Etapa IF**
  Durante la etapa de identificación IF, se realiza la misma identificación que se empleaba en las instrucciones tipo R, se identifica la operación y en caso de que se guarde sobre el banco de registros flotantes, se activa la variable de control pertinente. Además en el caso de las instrucciones de salto condicionadas por el registro especial se indica que este será el registro fuente que se empleara. Para de esta manera generalizar en la medida de lo posible la etapa de ejecución.

- **Etapa ID**
  En la etapa de decodificación, al margen de reconocer los registros empleados y recuperar sus datos resulta importante subrayar en este punto que los saltos tienen lugar en la etapa de decodificación es decir, se comprueba la condición de salto con los valores debidamente anticipados por el planificador, se calcula la dirección del salto y se devuelve la dirección que se recuperar en caso de que se tome o no el salto. En el simulador las instrucciones solo calculan la dirección de salto y recuperan los datos dado que el responsable de la condición de salto es el planificador.

- **Etapa EX**
  Dejando ya de lado las instrucciones tipo I de salto que ya han realizado todo lo que les correspondía, la etapa de ejecución EX realiza la operación correspondiente a la instrucción en la ALU y por lo tanto el comportamiento es el mismo que en las instrucciones tipo R sin embargo en este conjunto las operaciones de acceso a memoria emplean esta etapa para calcular la dirección que se accederá.

- **Etapa MEM**
  En la etapa de memoria solo realizan alguna operación las etapas de Store y Load. Llegada esta etapa si la instrucciones es de alguno de esos 2 conjuntos se recupera el dato accedido por la dirección si y solo si la dirección es accedida correctamente, en caso contrario se lanzara un error que parara el pipeline del simulador hasta una nueva compilación (esto también ocurre con las divisiones por 0 de la etapa ejecución). Si se desea acceder a una palabra se comprueba que la dirección es múltiplo de 4, si se accede a una media palabra, halfword se verifica que sea la dirección múltiplo de 2 , y si el acceso es a un valor exigido como flotante de doble precisión la dirección deberá ser múltiplo de 8.

- **Etapa WB**
  Por último a lo largo de la etapa de reescritura se guardan los resultados obtenidos en ejecución o accedidos desde memoria para su carga. Para ello las instrucciones tipo I al igual que hacían las tipo R simplemente devuelven el valor contenido en el atributo solución correspondiente de la instancia de "Instrucción".

4.3.5 Tipo J

Este pequeño conjunto de instrucciones solo está formado por instrucciones de salto.

Figura 31: Clase TipoJ.

Por ello el tratamiento que se da solo implica las dos primeras etapas, identificación de la operación como salto, y decodificación donde se calcula la dirección de salto si fuese necesario.
La característica principal de los saltos de este conjunto es que son todos incondicionales por lo tanto siempre tendrá lugar el salto a la dirección que acompaña la instrucción.
Asi pues en la etapa de identificación se reconoce la instrucción y en la etapa de decodificación se devuelve la dirección a la que se debe saltar.

Existe aun así una etapa de reescritura WB para alguna de estas se conserva el valor de PC o el valor de PC incrementado en alguno de los registros del banco de registros dado que dicha acción solo puede tener lugar en la etapa WB.

4.3.6 Tipo I Multimedia

El conjunto Tipo I Multimedia está formado todas por las instrucciones que emplean para llevar a cabo su ejecución algún tipo de inmediato, que suele oscilar entre un inmediato de 5 a 2 bits, con y sin signo.
Principalmente las acciones realizadas mediante estas instrucciones consisten en desplazar tantos bits como indique el inmediato o recuperar ese subconjunto de bits de un registro multimedia y replicarlo a lo largo de todo el conjunto. La clave de esta clase rige en el hecho de que emplean un inmediato. (Ver detalles en Apendice III).

Además de las instrucciones descritas anteriormente se deben añadir a este conjunto las instrucciones de acceso a memoria para registros multimedia, estas operaciones de transferencia de datos no emplean un inmediato como tal, sin embargo, su segundo registro fuente es un registro entero (tipo R) cuyo valor es recuperado como entero y empleado como inmediato para calcular la dirección accedida.

Figura 32: Clase TipoImultimedia.

Etapas

- **Etapa IF**
  Dado que el repertorio de instrucciones multimedia resulta más genérico que el repertorio DLX la etapa de identificación es la misma para todas las instrucciones sin ningún caso particular.

Durante esta etapa la clase responsable de este tipo, se identifica la instrucción que accede al pipeline y se indica el número de ejecuciones que tendrán lugar.

- **Etapa ID**

  Respecto a la etapa de decodificación ID no hay cambiado nada en relación a las tipo R e I, simplemente se actualiza la información de la instrucciones con los registros que se emplean y los inmediatos y con los datos correspondientes.

- **Etapa EX**

  La etapa de ejecución sigue con la misma dinámica que las instrucciones tipo I de acceso a memoria y las de ejecución de tipo R, solo que para el caso de las multimedia las operaciones implementadas poseen una mayor complejidad y dificultad al tener que fragmentarse 128 bits en los grupos que la instrucción indique. Durante esta etapa se emplean las conversiones para fragmentar los grupos descritas en conversiones (ver conversiones). Y por partes se lleva a cabo la operación correspondiente. Para las instrucciones de acceso a memoria se emplea esta etapa para calcular la dirección de acceso, a diferencia de las operaciones DLX de acceso, las operaciones Multimedia truncan el resultado para que la dirección sea múltiplo de 16.

- **Etapa MEM**

  La etapa de memoria solo afecta a las instrucciones descritas anteriormente que acceden a memoria para carga y almacenamiento de datos. Al haber ajustado al múltiplo de 16 inferior la dirección de memoria solo podrá pararse la ejecución como ocurría con las instrucciones tipo I si se trata de acceder a posiciones de memoria de fuera del rango de memoria (R[0..8191]), avisando del error al usuario. Al acceder a memoria se carga o se almacena en esa dirección de memoria y las 3 siguientes dado que el tamaño de palabra es de 32 bits y los registros multimedia son de 128 bits, o bien se debe fragmentar en 4 partes para su almacenamiento o bien se agrupan 4 palabras para formar el resultado que se guardara en el registro multimedia.

- **Etapa WB**

  Para finalizar en la etapa de reescritura WB, el comportamiento es idéntico a las instrucciones tipo R, se devuelve el resultado obtenido de la ejecución o del acceso a memoria (salvo para las operaciones de guardado en memoria).

4.3.7 Tipo R Multimedia

El conjunto de instrucciones pertenecientes a tipo R Multimedia son el resto de instrucciones multimedia no comentadas anteriormente.
No se realiza ninguna distinción particular entre todas ellas dado que para todas las etapas salvo la ejecución el comportamiento es el mismo, y para la etapa de ejecución el comportamiento que lo distingue es el fruto de la instrucción que las distingue.

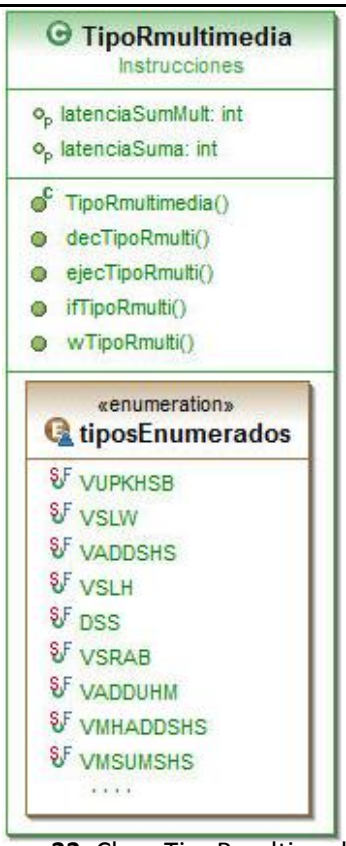


Figura 33: Clase TipoRmultimedia.

Resumiendo las etapas de identificación, reconocen la instrucción, durante la decodificación recuperan los datos e identifican los registros, en memoria ninguna realiza acción ninguna y durante la reescritura se devuelve el valor obtenido de la etapa de ejecución.
La etapa de ejecución para las multimedia requiere siempre fragmentar los datos de origen a tamaños de 32 bits, 16 bits, o 8 bits.
Una vez fragmentados se llevan a cabo las operaciones entre los distintos fragmentos de cada operando obteniendo un resultado partido que deberá reagruparse para formar un dato de 128 bits.

La unidad funcional multimedia es la misma que la empleada por las instrucciones DLX enteras, es decir, si se tratase de un procesador real, se habría tenido que modificar este, para emplear 128 bits (recordamos que en este proyecto solo se simula). Además, las instrucciones multimedia flotantes, emplean las mismas unidades funcionales que las instrucciones DLX flotantes, implicando una modificación similar a la comentada anteriormente.
Simulamos pues, que se ha aplicado una mejora en las ALU existentes en el DLX, para soportar operandos de 128 bits, las instrucciones DLX operarán con 32 bits mientras que las multimedia emplean 128 bits, por lo tanto las ALUs que se compartan con multimedia soportaran 128 bits.

4.4 Módulo Funcionamiento:

Índice:

4.4.1 Frame DLX

La clase “FrameDLX” es la responsable de unir todos los módulos y componentes del simulador, es la clase principal que contiene el “main” del programa.

Figura 34: Clase FrameDLX.

Aparte de incluir los componentes es la responsable de mostrar todos los datos y actualizaciones que se dan a lo largo de la ejecución.
Por ello esta es la que distribuye los elementos visualizados (memoria de datos, pipeline, banco de registros y código compilable) como se pueden apreciar en el manual de uso (Capitulo 5).

FrameDLX actualiza los datos referentes a la memoria de datos y al banco de datos pero únicamente de manera visual, recupera los datos que han sido modificados en los componentes respectivos y visualiza el nuevo dato.

En cuanto al trato del pipeline, esta clase no es crucial pero si importante, ya que al ojo del usuario es la responsable de mostrarle la progresión de la ejecución de las instrucciones.
Muestra a la derecha las instrucciones que se han identificado hasta ese punto de ejecución y a su izquierda en la misma fila muestra sus estados en el pipeline en función del ciclo en que se encuentre la ejecución.

Por último la barra de menú se gestiona mediante esta clase, permitiendo la carga de archivos y guardado exclusivamente con extensión ".s", abriendo la ventana de configuración y dando acceso a la ayuda en formato "pdf" contenida en la herramienta.

4.4.2 Planificación

El planificador resulta ser el componente más importante en el simulador DLX dado que será el encargado de controlar el correcto comportamiento de la maquina DLX.

El planificador es el responsable del control de riesgos, la anticipación y de la evolución correcta del pipeline del simulador DLX. Para llevar a cabo todo esto, disponemos de una clase "Planificación" que implementara el planificador para realizar las tareas anteriormente comentadas en este proyecto.

La planificación seguirá el funcionamiento de una maquina de estados, en concreto, dispondrá de un listado de instrucciones que podrán encontrase en 5 estados distintos que representan las etapas básicas de identificación (IF), decodificación (ID), ejecución (EX), memoria (MEM) y reescritura (WB) cada instrucción podrá encontrarse en estados de parada.

Figura 35: Clase Planificación.

Comportamiento general y observaciones

Una vez por ciclo, el planificador ejecuta su función principal, en la cual se recorren una tras otra todas las instrucciones por orden de antigüedad de acceso al pipeline. El funcionamiento genérico del planificador consiste en una por una recorrer las instrucciones, y tratarlas según su estado (a continuación veremos los estados).
La clave del planificador es que cada instrucción que trata la deja en la etapa que deberá realizar en el próximo ciclo para tratarlo como una máquina de estados. Por ello añade siempre una instrucción por delante del ciclo en el que se encuentra, para dejarla en su etapa IF. Esto ocurre en todas las iteraciones salvo en la primera, que se lee la primera instrucción se hace ejecuta su etapa de identificación y se añade la siguiente instrucción para preparar su ejecución en el siguiente ciclo. Por ello se debe distinguir un pipeline vacio porque ya termino o un pipeline vacio de comienzo de código.

Cuando una instrucción entra en el pipeline lo primero será comprobar su tipo (R, I, J, RMultimedia e IMultimedia) dado que en función de este el trato de cada etapa será diferente.
Para dejar correctamente la instrucción lista para el siguiente ciclo, al final de cada tratamiento de instrucción de comprobará si hay o no parada del pipeline mediante la unidad detección de riesgos.

Maquina de estados del planificador

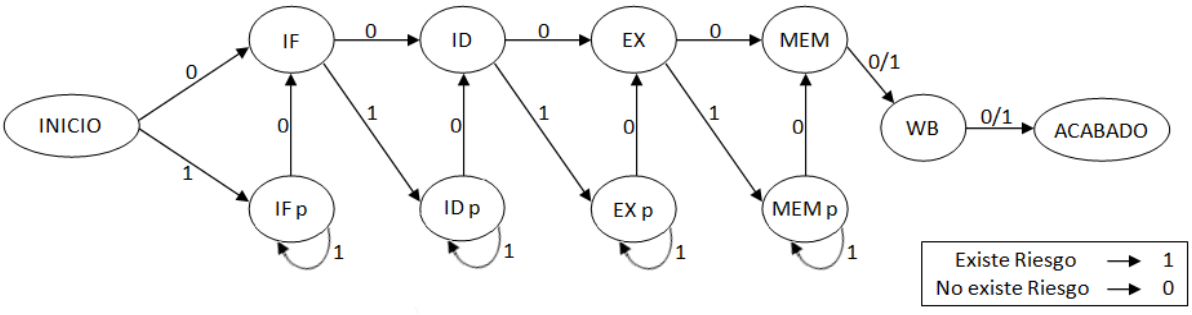


Figura 36: Diagrama de estados.

Estados

- **Inicio**
  El estado "inicio" existe para preparar la instrucción que en el próximo ciclo entrará en el pipeline que según la unidad de detección de riesgos pasará al estado IF o IF parada.
  De esta manera el listado de instrucciones del pipeline incluirá una instrucción extra respecto a lo que ocurriría en una máquina real, pero ésta no se visualizará, simulando la visualización del pipeline bajo el estado esperado.

  Dicho estado es necesario ya que la etapa que se representa en código de una instrucción es la siguiente a la que se visualiza. Por ejemplo: Si la etapa de una instrucción es IF, significa que aún tiene que pasar por la etapa correspondiente a IF, una vez indique que su etapa es ID será porque la instrucción está preparada para su futura etapa. Por ello es necesario introducir una instrucción al pipeline codificado cuya etapa será IF pero sin ejecutarla por esa unidad.

- **IF p**
  El estado "IF parada" se debe a que la unidad de detección de riesgos en el estado inicio detecto que la unidad para realizar la identificación se encontraba ocupada por otra instrucción anterior. La instrucción está cargada en el pipeline aunque no haya ejecutado su identificación. Si ocurre que alguna instrucción se encuentra en esta etapa el pipeline quedará parado al no poder acceder ninguna nueva instrucción a la máquina de esta manera la instrucción cuya etapa es IF p será al ultima instrucción del listado en ejecución.

En este estado se consultará de nuevo la unidad de detección de riesgos para marcar si la instrucción se encontrará en el siguiente ciclo en la etapa IF p si existe un riesgo o permitir el acceso a la unidad de Identificación modificando su etapa a IF.

- **IF**

  El estado "IF", se encargara de la identificación. En el simulador la etapa IF identifica la operación anotándola en la tabla del pipeline. A continuación se consultará de nuevo la unidad de detección de riesgos comprobar si puede pasar a la etapa ID o deberá permanecer en el buffer de IF marcando su etapa como ID parada.

  En esta etapa además se incrementara el contador de programa, en el simulador esta anotado como "paso" para poder identificar a la siguiente instrucción que debe prepararse. Se ha simulado un multiplexor que selecciona si se realiza o no el salto. Si se realizase el salto se pondría el contador de programa a la dirección de salto, si no se tomaría el valor incrementado del PC.

- **ID p**

  El estado "ID parada" puede ocurrir por dos razones, o bien la unidad de decodificación se encuentra ocupada por otra instrucción del pipeline o bien porque existe una dependencia de datos (ver explicación Unidad de riesgos). Esta etapa no realiza ninguna acción especial, simplemente consultara la unidad de detección de riesgos para actualizar la instrucción a ID o mantenerla en su estado actual.

- **ID**

  El estado "ID" representa la decodificación de la instrucción que se encuentra en la unidad. A nivel de nuestro simulador, se actualizará la información visualizada de la instrucción en la tabla del pipeline con los datos de los registros fuente y destino accedidos por la instrucción. Además se extraerán los datos necesarios del banco de registros para llevar a cabo la operación que tendrá lugar en la etapa de ejecución o acceso memoria y se le añadirán a la instancia de "Instrucción" que contiene la información de la instrucción que se encuentre en la etapa ID.

  En el caso de los saltos, si una instrucción de salto se encuentra en esta etapa, se evaluará la condición de salto y se calculará la dirección de salto. Si fuera necesario se anticiparían los operandos necesarios a la etapa de decodificación.

Por último y como en todas las etapas, se consultara la unidad de detección de riesgos modificando la etapa de la instrucción para indicar en el próximo ciclo que etapa deberá llevarse a cabo, EX p, EX 1, o EX.

- **EX p**

  El estado "EX parada" para no realiza ninguna operación sobre la instrucción, se encuentra en este estado porque la unidad funcional de la operación que corresponda a la instrucción se encuentra ocupada, ya sea la parte segmentada a la que se desea acceder o a la unidad no fragmentada entera. Se comprueba si existe algún riesgo y si no existe se modifica la etapa de la instrucción a EX "y", si la etapa de ejecución se compone varias subetapas, donde "y" es el número de la subetapa de ejecución, en el caso de las operaciones sobre flotantes. O a la etapa EX si solo existe una etapa para la ejecución, es el caso de las operaciones sobre enteros o enteros multimedia. Para el simulador, se anticipan los datos en ejecución parada si fuese necesario (esto se debe a la mejora de inhibición de escritura).

- **EX**

  El estado "EX" realiza la operación requerida por la instrucción con los datos extraídos de los operandos fuente, y guardando, en la instancia "Instrucción" de la instrucción, el resultado obtenido para escribirlo en la etapa WB en el banco de registros. Este estado solo existe para todas las operaciones que no realicen aritmética con flotantes. Se consulta si existe dependencia y de ser así se anticiparían los datos que dependen, de las instrucciones anteriores. Finalmente consulta si la unidad de memoria está disponible para actualizar su etapa a MEM. Desde esta etapa se puede anticipar los datos a las instrucciones que dependan de la instrucción que acabe de finalizar su etapa de ejecución.

- **EX 1**

  El estado "EX 1" consulta si existe dependencia y de ser así se anticiparían los datos que dependen, de las instrucciones anteriores para simular la máquina DLX. Este estado solo aparece en las instrucciones aritméticas en coma flotante. Comprueba en la unidad de riesgos si puede avanzar a su siguiente etapa

- **EX f (f es la última subetapa de ejecución)**

  El estado "EX f" se comporta como el estado "EX", se realiza la operación de la instrucción con los datos extraídos de los operandos fuente, y guardando, en la instancia "Instrucción" de la instrucción, el resultado obtenido para escribirlo en la etapa WB en el banco de registros.

Este estado solo existe para todas las operaciones que realicen aritmética con flotantes. Ya no se comprueba la anticipación los datos correctos ya fueron tomados en la etapa EX 1. Una vez realizada la operación recupera la información de si existe riesgo o no para acceder a la etapa MEM en el próximo ciclo. Desde esta etapa se puede anticipar los datos a las instrucciones que dependan de la instrucción que acabe de finalizar su etapa de ejecución.

- **EX [2..f-1]** (subetapas de ejecución de la segunda hasta la penúltima)
  No realizan nada simplemente, simulan el comportamiento de las unidades funcionales de ALU. Consultaran la unidad de riesgos para saber si pueden o no pasar a su siguiente etapa de ejecución.

Cuando una instrucción entra en cualquiera de las subetapas de Ejecución, se marca como ocupada la posición en la unidad funcional correspondiente a dicha subetapa. Y por lo tanto al pasar a la siguiente etapa liberan la posición que ocupaban en la unidad funcional.
El número de subetapas es el mismo que el número de latencia de las unidades funcionales y por lo tanto es el número que introdujo el usuario en la configuración de la ALU antes de compilar el código que se va a ejecutar.

- **MEM p**
  El estado "MEM parada" ocurre exclusivamente cuando una instrucción ha terminado su o sus etapas de ejecución y el acceso a memoria se encuentra ocupado por otra instrucción anterior en el pipeline. Esta etapa tiene lugar generalmente cuando dos instrucciones terminan su etapa de ejecución a la vez y no comparten unidad funcional (sino no se daría el caso), de esta manera la instrucción que lleva más tiempo en el pipeline pasa a tomar el control de la unidad de memoria dejando en MEM p a la otra. Se llama al control de riesgos para saber si podremos acceder a memoria en el próximo y ciclo y por lo tanto la instrucción pasa a la etapa MEM.

- **MEM**
  El estado "MEM", será el responsable del acceso a memoria por parte de las instrucciones, esto solo se da en las instrucciones de carga y almacenamiento, el resto pasan por la memoria pero sin hacer nada. Sin embargo todas pasan por esta etapa antes de pasar al WB. Desde esta etapa se puede anticipar al igual que desde las etapas de ejecución finales, los datos mediante cortocircuito para evitar las paradas de dependencia en el pipeline. A diferencia de todos los estados vistos anteriormente en éste no se consulta la unidad de detección de riesgos ya que no es posible que 2 instrucciones deseen

utilizar la unidad de reescritura WB a la vez. Esto se debe a que para que esto ocurriese las 2 instrucciones deberían encontrarse en su etapa de memoria, imposible dada la naturaleza del pipeline del DLX.

Si nos encontramos con algún tipo de instrucción Store, la tabla de la memoria se actualiza.

- **WB**
  El estado "WB" se encarga de almacenar los resultados en el banco de registros. Para ello cuando una instrucción se encuentra en su etapa de reescritura el resultado de la operación, de la ALU o de acceso a memoria se guardara en el registro destino.

  Puede ocurrir que el dato no se almacene, en ese caso se deberá a la inhibición de escritura, se comprueba si la instrucción debe escribir o no, con una variable contenida en la instancia de "Instrucción" que la representa, que actualizó correctamente la unidad de riesgos, si no debería seria porque el registro destino ya ha sido actualizado con un dato más actual.

Descripción de los atributos del Planificador

Para cumplir con el comportamiento especificado mediante la máquina de estados. La clase "Planificación" dispondrá de los siguientes atributos.

- **Instrucciones:**
  En primer lugar un conjunto de instrucciones auxiliares de los cinco tipos tratados en el simulador, R, I, J, RMultimedia e IMultimedia. La finalidad de estos atributos es indexar las instrucciones que están recorriendo el pipeline, para un uso de programación más sencillo. Son instancias de las clases que llevan los métodos para tratar las instrucciones en su etapa y no instancias de "Instrucción" que son lo que denominamos instrucciones a lo largo del planificador.

- **Atributos graficas:**
  Una instrucción únicamente será eliminada del pipeline, cuando haya finalizado su etapa de Write-Back. Para ello necesitaremos guardar en cada ciclo, la instrucción (si la hubiera) que al final de este tendrá que ser eliminada.
  Por motivos de representación gráfica del pipeline en la tabla, es necesario guardar, en la instancia de dicha instrucción aparte de las del pipeline, la instrucción que acaba de finalizar su etapa WB, esta no pertenece al pipeline ya pero se necesitan sus datos para la visualización simulada.

- **El Pipeline de instrucciones:**
  Sera necesario también y es el elemento más importante del planificador, un vector de instancias de "Instrucción", que contendrá las instrucciones que en ese ciclo están dentro del pipeline y deberán ser tratadas en función de su estado. Será, en cierto modo, lo que simule el contenido de los registros entre etapas del DLX.

- **Contador de programa y control de salto:**
  Dado que el planificador controla el pipeline será necesario saber el paso en que se encuentra el contador de programa, en nuestro caso se incrementa de uno en uno simplemente para indexar a la siguiente instrucción que le corresponde el acceso al pipeline. Simulamos el paso mostrando el contador de programa, que es el paso*4.
  Se lleva también el control del paso anterior para la representación gráfica, para saber si se ha tomado salto o no en el paso actual.
  Es necesaria una variable para saber si debemos o no tomar el salto.

  Por motivos de control se requiere una variable que nos indique si la instrucción que estamos tratando es la ultima instrucción de todo el código compilado.

- **Estadísticas de la ejecución:**
  Dado que en el pipeline es donde sabemos que instrucciones saltos, riesgos, y demás características del DLX tiene lugar el planificador contara con una instancia de "estadísticas" para ofrecer la posibilidad de mostrarle al usuario los datos extraídos de la ejecución de un programa.

- **Control Unidades Funcionales:**
  Se requerirá un booleano que indique si la unidad funcionar de la división flotante está o no en uso, dado que al no estar segmentada, solo una instrucción de división puede entrar en ejecución a la vez. Si otra instrucción de división quiere entrar en ejecución, será necesario que espere a que la primera haga su etapa de Memoria, es decir, que libere la unidad funcional de división no segmentada.

Finalmente en relación a la etapa de ejecución el planificador tendrá una instancia de "UnidadFuncionalFlotante" para saber que unidades flotantes segmentadas están disponibles a través de la unidad de detección de riesgos.
Asociada a esta unidad será necesario llevar el numero en el que esta segmentada cada unidad funcional existente que tendrá el valor que el usuario desee.

4.4.3 Tratamiento de BreakPoints

Para ampliar la funcionalidad del simulador, se opta por introducir una mejora en la depuración paso a paso. Dicha mejora consiste en la posibilidad de introducir breakpoints (puntos de interrupción) a la hora de ejecutar el código compilado.

Figura 37: Clase TratamientoBreakPoints

Esta mejora permite ejecutar el código hasta la instrucción que usuario desee sin la necesidad de ir paso a paso.
Para llevar a cabo tal mejora será necesario disponer de una clase que nos permita almacenar un listado de índices, es decir de breakpoints, donde el código deberá detener su ejecución. Además esta clase contará el número de líneas que existen con código compilable.

La clase identifica la línea que seleccionó el usuario y si esta se encuentra posterior a la directiva .text se añade al listado de puntos de parada. La clase se encarga de detectar si en el punto señalado existe o no texto y de ser así obtiene el índice de la instrucción señalada por el usuario, el cual se añadirá al listado.

Finalmente a la hora de ejecutar el código en el pipeline se irá comprobando si existe algún breakpoint y si se ha alcanzado la instrucción que referencia para detener la ejecución permitiendo su futura reanudación.

4.4.4 Configuración

La configuración de la que disponemos en el simulador, es una pieza que aumenta notablemente la eficiencia del simulador como herramienta docente.
Distintas pestañas nos permiten acceder a todas las opciones de configuración. O mediante un menú desplegable al pulsar botón derecho sobre el bando de registros o memoria

Figura 38: Clase Configuración.

- **ALU:**
  Mediante la configuración que se muestra en una ventana extra el usuario podrá modificar y alternar el tamaño de las distintas unidades funcionales correspondientes a la etapa de ejecución. Solo se pueden modificar las unidades funcionales en coma flotante, las de enteros permanecerán intactas. Con el tamaño nos referimos al número de ciclos de latencia que lleva la ejecución.
  Por defecto Suma/Resta tiene 4 ciclos de latencia, la multiplicación 7 ciclos, la división que no se encuentra segmentada (el resto si) son 24 ciclos y por ultimo Suma/Resta con multiplicación, solo empleada por las instrucciones multimedia tiene una latencia de 7 ciclos.

- **Memoria de datos:**
  La configuración permite en todo momento cambiar el formato en que se están visualizando los datos contenidos en la memoria de datos. Para mejorar la herramienta se pone a disposición la visualización en formato, binario (por defecto), decimal (cada palabra se visualiza como un entero), coma flotante (cada palabra es un flotante de simple precisión y hexadecimal.

- **Banco de Registros:**
  Al igual que para la memoria, la configuración ofrece la posibilidad de visualizar los mismos formatos antes comentados, con algunos añadidos. Al tratarse del banco de registros y por lo tanto de los datos con los que se lleva a cabo las ejecuciones, toca diferenciar distintos formatos en función del tipo de registro.

Añadiendo así la opción muy agradable a la hora de operar con los datos multimedia de mostrar los datos agrupados por tamaño de byte, halfword y word.
Los registros flotantes permitiendo su visualización como formato de doble precisión. Y como ya se ha comentado todos los tipos de registros, enteros, flotantes y multimedia, permiten su visualización en los formatos, decimal, coma flotante, binario y hexadecimal.

- **General:**
  Como configuración general, y para proseguir con la mejora de la herramienta didáctica, se puede configurar el idioma entre, inglés, francés y español.

4.5 Problemas encontrados

Índice:

4.5.1 Anticipación entre registros

A la hora de hacer la anticipación entre datos tuvimos bastantes problemas, ya que había que recorrer todas las instrucciones que iban por detrás en el pipeline, para comprobar si existía dependencia. Además de eso, había que comprobar, que si una instrucción dependía de dos de las que iban antes que esta, la dependencia debía efectuarse con la más cercana a ella. También tuvimos que comprobar que la instrucción de la que intentabas anticipar el dato, hubiera llegado a una etapa en la que ya tuviera listo el dato, es decir, la última EX, excepto en las instrucciones de carga, que el dato queda listo en la instrucción de memoria.
En caso de que los datos fueran en punto flotante (64bits, double), buscar la instrucción de la que podías depender era más complicado, ya que había que comprobar el registro par y el sucesivo impar. Tanto si la instrucción double era de la que dependías, como si la double era la que necesitaba coger el dato de otra.

4.5.2 Introducción de las instrucciones multimedia (ALTIVEC)

Cuando habíamos terminado de implementar toda la parte del procesador DLX simple, decidimos introducir el repertorio de instrucciones multimedia, lo cual produjo que tuviéramos que cambiar bastantes cosas en la implementación. Entre otras cosas, hubo que cambiar el banco de registros, ya que estas instrucciones tenían un tamaño de 128 bits (cuatro veces más que el tamaño de las instrucciones del DLX). Hubo que cambiar las dependencias, ya que hay instrucciones multimedia que dependen de instrucciones DLX, por lo que las dependencias de las multimedia no eran aisladas a las dependencias que teníamos anteriormente.

4.5.3 Gráfico

Al principio, el gráfico que habíamos propuesto no permitía que se ejecutasen las instrucciones flotantes (ni las multimedia flotantes), ya que la etapa de ejecución duraba un ciclo de reloj. Por lo que hubo que cambiar el esquema para separar las instrucciones flotantes en una unidad que tardaba un número de ciclos variable, según la latencia introducida por el usuario para cada unidad funcional flotante.

Capítulo 5

Manual de Uso

Índice:

5.5 Menú Opciones
- Configuración
  - General
  - Banco de registros
  - Memoria
  - ALU
- Idioma
- Estadísticas

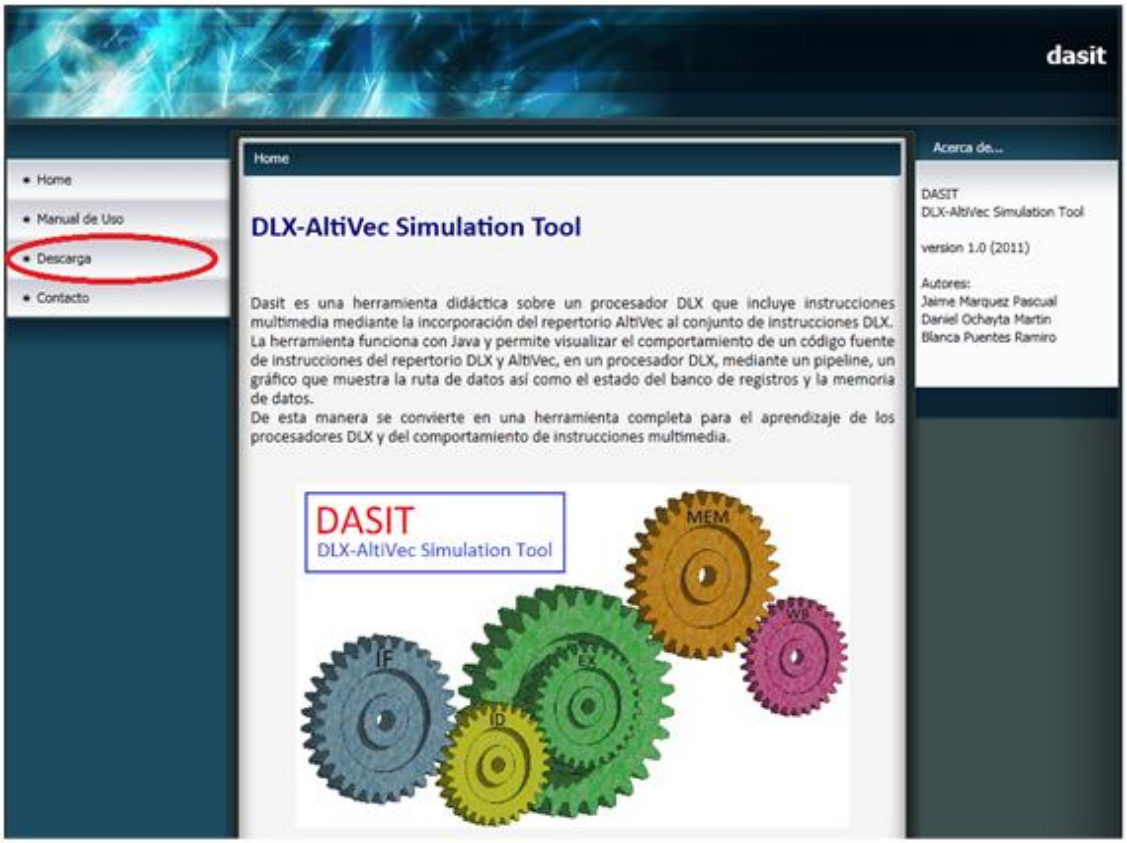
5.6 Menú Ayuda
- Directivas DLX
- Instrucciones DLX
- Instrucciones AltiVec
- Errores
- Manual de Uso
- Acerca de

5.7 Pipeline
5.8 Gráfico
5.9 Banco de Registros
5.10 Memoria de Datos
5.11 Área de código
5.12 Accesos Directos

5.i Descarga de la herramienta

La herramienta puede descargarse en el siguiente enlace:
http://dasit.es.tl/

Seleccionar la opción “Descarga” en el menú de la parte izquierda.

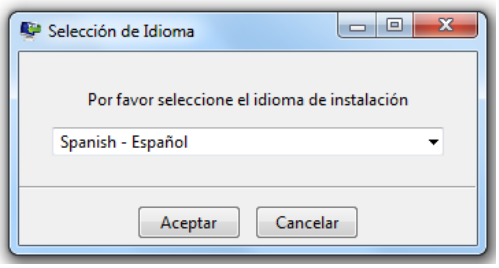


Se puede elegir la última versión de DASIT, cuyo enlace estará situado como primario. Pero también se podrán descargar versiones anteriores, marcadas como tales.

5.ii Instalación y Desinstalación

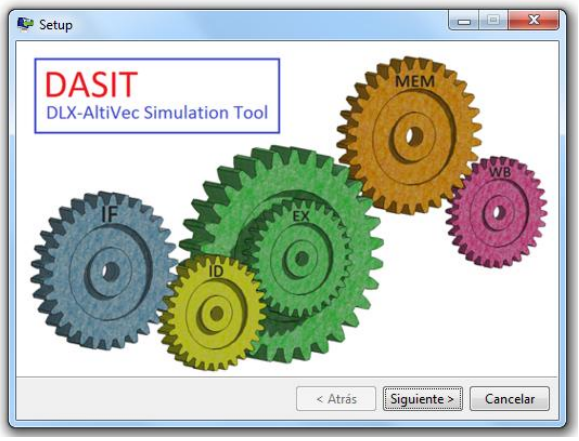
Paso 1:

Elegir el idioma en el que deseamos seguir la instalación.



Paso 2:

Para comenzar la configuración, pulsamos “Siguiente”.



Paso 3:

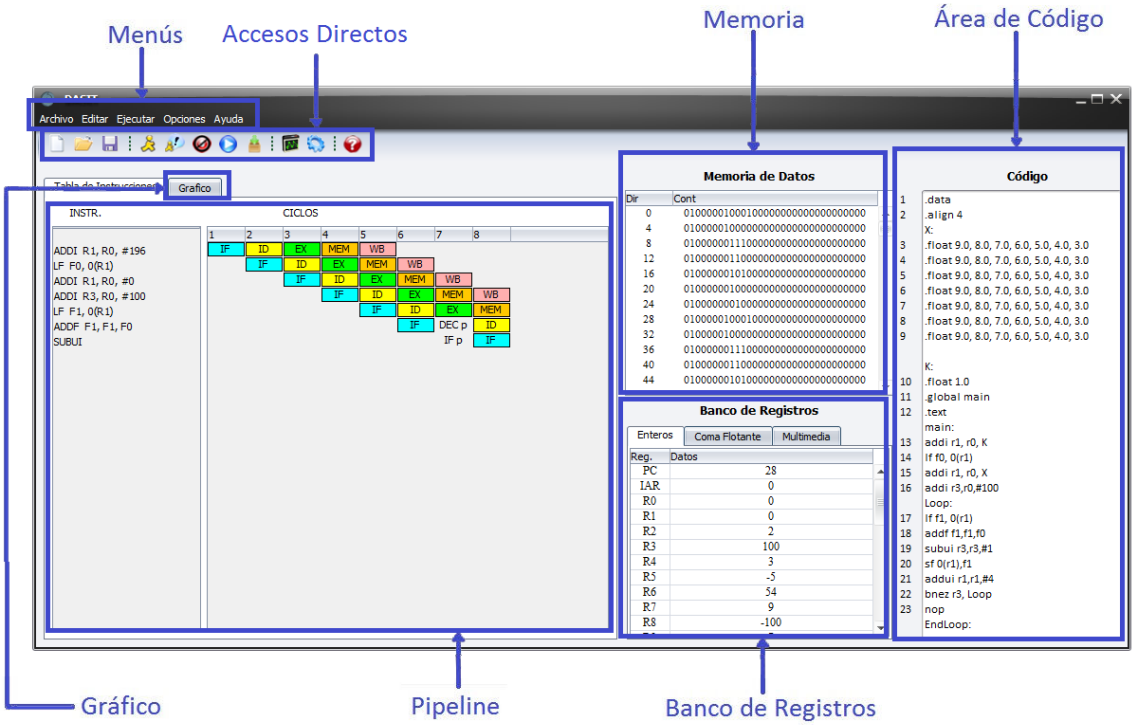
Seleccionamos el directorio donde deseamos que se instale la aplicación.

Una vez, el programa se haya instalado correctamente en la ruta marcada, se abrirá el archivo “README”, donde se muestran los detalles de la herramienta.

El programa se puede ejecutar mediante el archivo “DASIT.jar”.

5.1 Panel principal:

En el panel principal del reproductor se encuentran:

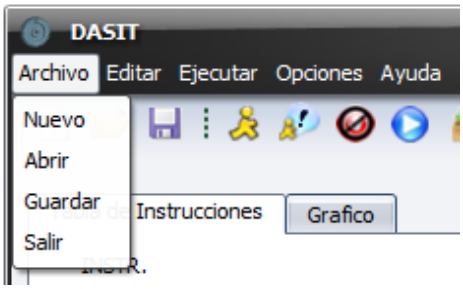


- Los **menús** “Archivo”, “Editar”, “Ejecutar”, “Opciones” y “Ayuda”, que incluyen las diversas funcionalidades del simulador.
- El **pipeline**, donde va mostrándose la evolución del código ejecutado.
- El **gráfico**, donde va mostrándose la evolución del código ejecutado pero desde el punto de vista del Hardware.
- La **memoria de datos**, donde se pueden visualizar los 8kbytes que reservamos para guardar los datos en memoria.
- El **banco de registros**, que se divide en tres zonas, para visualizar los 32 registros de cada tipo (enteros, flotantes y multimedia), así como sus registros especiales y el contador de programa.

- El **área de código**, donde el usuario introduce el código que quiere ejecutar.
- Los **accesos directos**, donde se ven las mismas opciones que los menús.

5.2 Menú Archivo:

Dentro del menú Archivo se encuentran las siguientes opciones:



Nuevo:

Borra el área de código y limpia el pipeline y el gráfico.

Abrir:

Al abrir un código guardado en un archivo con extensión ".s", este se mostrará en el área de código, pero por defecto, no se autocompilará, ni se borrara el pipeline si hubiera un código anterior. Puede configurarse.

Guardar:

Abre una ventana de diálogo en la que seleccionar una ubicación para el nuevo archivo de código. Una vez añadido un nombre para el archivo, después de Aceptar, se creara un archivo con la extensión ".s" en el lugar indicado. Si hubiéramos cargado anteriormente un código, este se guardara por defecto en el mismo archivo.

Salir:

Salir de la aplicación.

5.3 Menú Editar:

Dentro del menú Editar se pueden encuentrar las siguientes opciones:

Borrar Banco de Registros:

Limpia el banco de registros poniendo como valor un 0 (incluyendo los registros especiales).

Borrar Memoria:

Limpia todas las posiciones de memoria, insertándoles como valor un 0.

Insertar en Banco de Reg.:

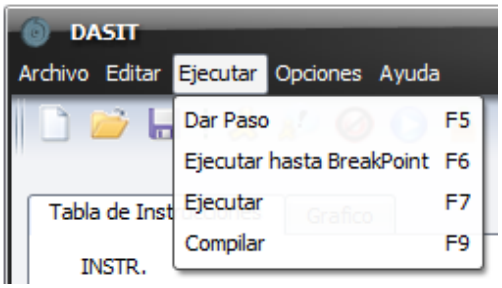
Esta opción solo funciona para los registros enteros y flotantes. Se abre un nuevo panel donde se pide al usuario que introduzca el número de registro que desea modificar, el tipo de registro y por último el dato correcto que desea introducir.

Insertar en Memoria:

Se abre un nuevo panel donde se pide al usuario que introduzca la dirección de memoria que desea modificar, el tipo de dato que va a introducir (entero o flotante) y por último el dato correcto que desea introducir.

5.4 Menú Ejecutar:

En el menú Ejecutar se ofrecen las posibilidades siguientes:



Dar Paso:

Muestra el resultado del siguiente ciclo de la ejecución, tanto en el pipeline como el gráfico.
Tiene como acceso directo la tecla F5.

Ejecutar hasta BreakPoint:

Si el usuario ha introducido algún punto de interrupción en el código, este se ejecuta hasta ese punto. En caso contrario, el código se ejecutará hasta el final.
Tiene como acceso directo la tecla F6.

Ejecutar:

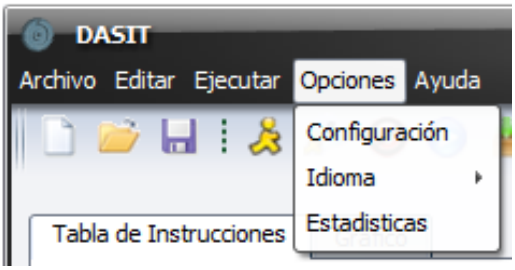
Muestra el resultado de la ejecución de todos los ciclos que dura el programa que hemos compilado, tanto en el pipeline como el gráfico. Durante esta ejecución se mostrara un nuevo panel en el que el usuario tiene la posibilidad de parar la ejecución en cualquier momento.
Tiene como acceso directo la tecla F7.
No se puede ejecutar un código si anteriormente no se ha compilado.

Compilar:

Se comprueba que el código introducido por el usuario no tiene errores, en caso de que hubiera alguno saldrá un aviso por pantalla. El código queda listo para su ejecución.
Tiene como acceso directo la tecla F9.

5.5 Menú Opciones:

En el menú Ejecutar se pueden ver las siguientes posibilidades:



Configuración:

Se abre un nuevo panel en el que el usuario tiene la posibilidad de configurar el simulador DLX.

General:
El usuario tiene la posibilidad de elegir el idioma en el que desea que este la aplicación. Se pueden seleccionar opciones como autocompilar y borrar el pipeline al abrir, así como la velocidad a la que queremos que vaya la "Ejecución".

Banco de registros:
Se puede seleccionar el formato en el que desea que se muestren los Bancos de Registros. En caso de que se muestren en formato numérico, podremos elegir si deseamos que la zona de registros multimedia se muestre en decimal o en coma flotante.
Existe la posibilidad de mostrar el banco de registros flotantes en formato double, es decir, se concatena cada registro par, con su sucesivo registro impar, para formar valores de 64 bits, en lugar de flotantes de simple precisión.
Además los registros multimedia podrán visualizarse divididos en Word (1 conjunto de 32bits), HalfWord (2 conjuntos de 16 bits) o byte (4 conjuntos de 8bits).

Memoria:
Se puede seleccionar el formato en el que desea que se muestre la Memoria.

ALU:
Donde se puede elegir los ciclos de latencia de las Unidades Funcionales de Coma Flotante que se desee. Los ciclos deberán ser números naturales, es decir, como mínimo puede haber un ciclo de latencia.
Estas modificaciones se tendrán en cuenta al compilar el código.

Idioma:

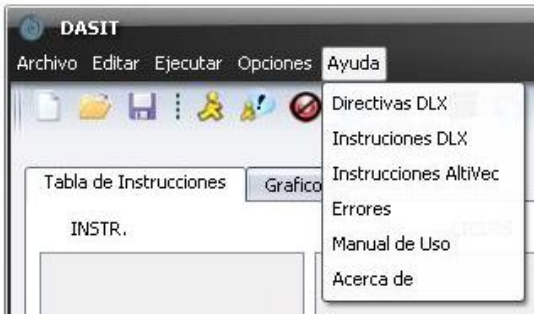
El usuario tiene la posibilidad de elegir el idioma en el que desea que este la aplicación. Los idiomas disponibles son Español (por defecto), Inglés y Francés.

Estadísticas

Muestra las estadísticas en cualquier momento de la ejecución. Podemos elegir que información deseamos que se muestre y ocultar la que no, pulsando sobre ellas con el ratón.

5.6 Menú Ayuda:

Dentro del menú Ayuda se encuentran las siguientes opciones:



Directivas DLX:

Abre automáticamente un pdf con el formato y la explicación de las directivas DLX que se pueden emplear en el código. Corresponde al apéndice II.

Instrucciones DLX:

Abre automáticamente un pdf con el formato y la explicación del repertorio de instrucciones DLX que se pueden emplear en el código. Corresponde al anexo I.

Instrucciones AltiVec:

Abre automáticamente un pdf con el formato y la explicación del repertorio de instrucciones multimedia AltiVec que se pueden emplear en el código. Corresponde al anexo III.

Errores:

Abre automáticamente un pdf con la explicación de los errores que pueden darse al compilar un código. Corresponde a los errores descritos en el compilador (Capitulo 4).

Manual de Uso:

Abre automáticamente un pdf con todo el manual de uso aquí explicado.

Acerca de:

Muestra el nombre, la versión y los créditos del programa.

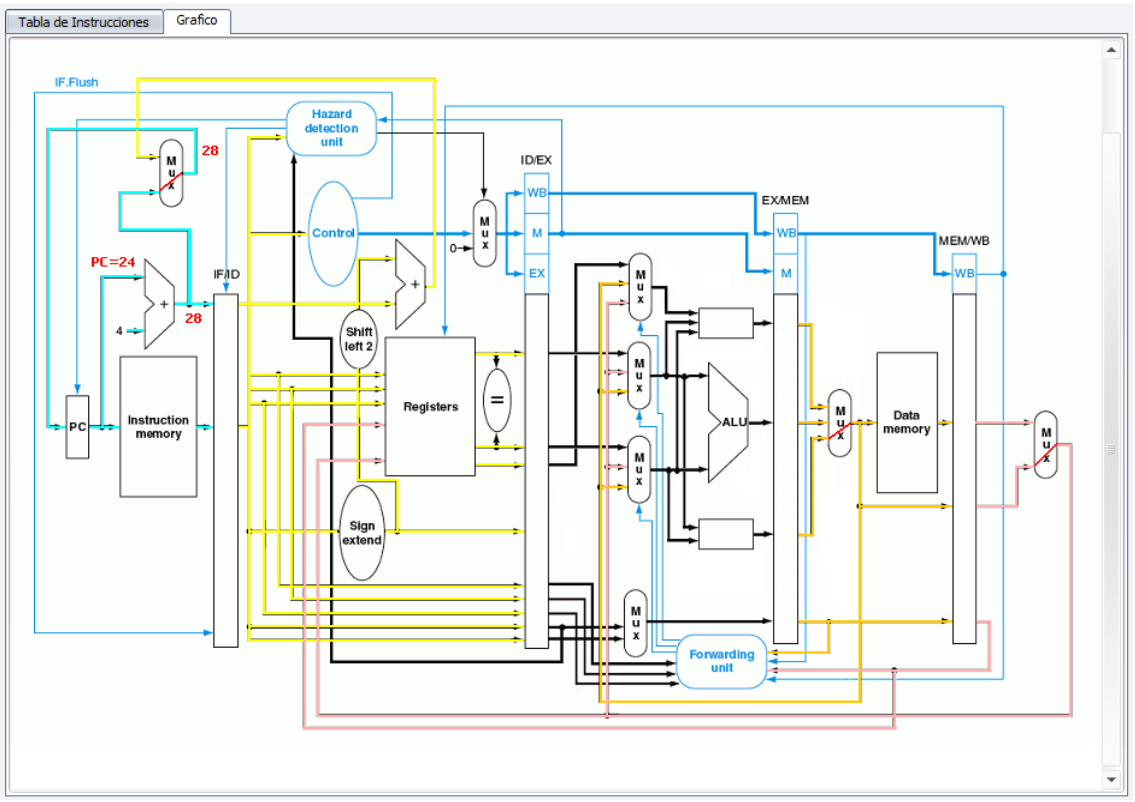
5.7 Pipeline:

En este campo van saliendo las instrucciones según el ciclo en el que se van ejecutando, de esta manera, se puede ver en qué etapa esta cada instrucción del pipeline en un ciclo determinado.

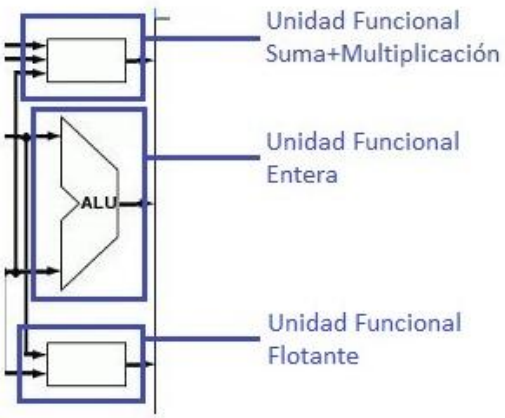
El usuario no puede interactuar con el pipeline.

5.8 Gráfico:

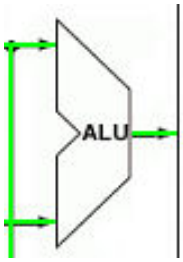
En este campo se pueden visualizar las instrucciones desde el punto de vista del hardware.



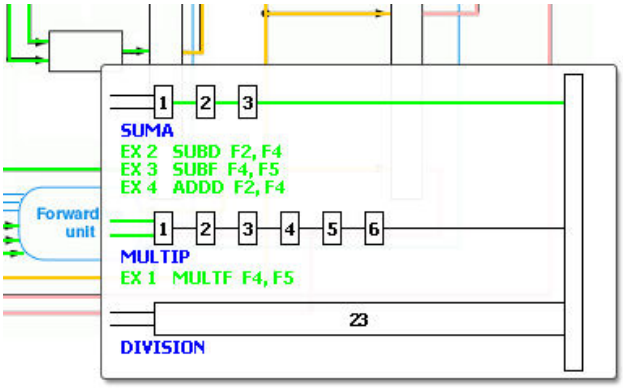
La etapa de ejecución, se puede dividir en tres partes, ya que hay tres tipos de Unidades Funcionales.



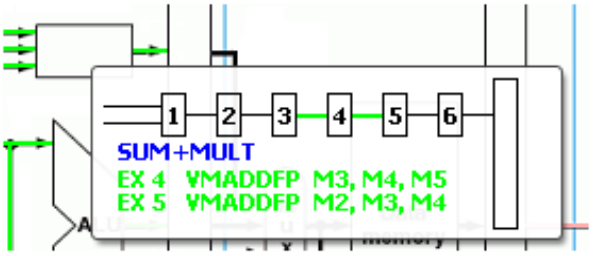
Para las instrucciones enteras, se utiliza la ALU del procesador DLX simple, ya que estas instrucciones siempre tienen un ciclo de latencia en la ejecución, y no se puede modificar en ningún momento.



Para las instrucciones flotantes, se necesita una Unidad Funcional que opere en coma flotante, ya que la latencia de la ejecución de estas instrucciones puede ser variable.
Para poder visualizar los diferentes ciclos de la ejecución de estas instrucciones, basta con pasar el ratón por encima de de dicha Unidad Funcional.

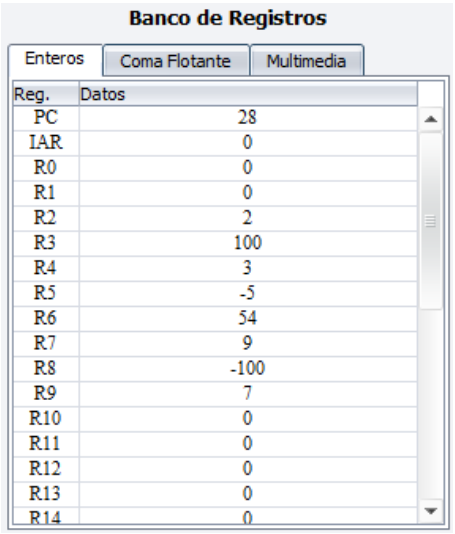


Para las instrucciones multimedia que necesitan hacer una operación de multiplicación precedida de una operación de suma, se necesita una Unidad Funcional que combine estas dos operaciones. Además la latencia de la ejecución de estas instrucciones puede ser variable.
Para poder visualizar los diferentes ciclos de la ejecución de estas instrucciones, basta con pasar el ratón por encima de de dicha Unidad Funcional.



5.9 Banco de Registros:

El banco de registros esta se dividido en tres zonas, 32 registros enteros, 32 registros flotantes y 32 registros multimedia. Además existe un registro especial para cada tipo de registros.



Los registros enteros siempre son valores enteros de 32bits. Los registros flotantes también son valores expresados en coma flotante de 32 bits, pero el usuario tiene la posibilidad de visualizarlos como doublés, por lo que cada registro par es concatenado con su registro sucesor impar, formando 16 registros de 64 bits cada uno. Los registros multimedia son valores de 128bits, pero los visualizamos en cuatro grupos de 32 bits, ya que cuando se opera con instrucciones multimedia nunca se toman los 128bits juntos.

En este campo, también podremos visualizar el contador de programa (PC).
Se marca en azul claro el último registro modificado.

Usando el botón secundario del ratón se abre un menú en el que podemos seleccionar los siguientes campos:

Vista: Como queremos que se muestren los valores de la memoria. Podemos elegir entre “Binario”, “Decimal”, “Coma Flotante” y “Hexadecimal”. Exceptuando que los registros enteros nunca podrán visualizarse en formato “Coma Flotante”, y a su vez, los registros en coma flotante nunca podrán visualizarse en formato “Decimal”.

Copiar: Guarda los 32bits seleccionados.

Pegar: Carga en el registro seleccionado los 32bits que anteriormente copiados.
Podemos copiar y pegar valores desde unos registros a otros (aunque sean de diferentes tipos), ya que se copiaran los bits, no los valores en un determinado formato.

Modificar Valor: Abre un nuevo panel en el que se le pide al usuario que introduzca un valor entero o en coma flotante, según el registro que hayamos seleccionado, para introducírselo a dicho registro.

Borrar Valor: Pone los 32bits del registro seleccionado a 0.

Limpiar Banco de Reg.: Borra todas los registros, poniendo a 0 todos los bits.

Además en los Registros en Coma flotante, tendremos la opción de visualizar los registros en formato doublé o no.
En los registros Multimedia, el usuario tendrá la opción de separar los bits en tres formas:

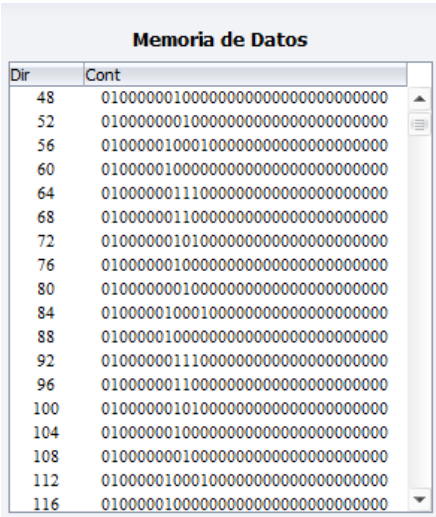
- Word (1 conjunto de 32bits)
- HalfWord (2 conjuntos de 16 bits)
- Byte (4 conjuntos de 8bits)

5.10 Memoria de Datos:

La memoria de datos de este Simulador ocupa 8kbytes. Y se muestra en grupos de 32 bits. Se marca en azul claro la última dirección de memoria modificada.

La memoria de datos de este Simulador ocupa 8kbytes. Y se muestra en grupos de 32 bits.

Se marca en azul claro la última dirección de memoria modificada.



Usando el botón secundario del ratón se abre un menú en el que podemos seleccionar los siguientes campos:

Vista: Como queremos que se muestren los valores de la memoria. Podemos elegir entre “Binario”, “Numérico”, “Coma Flotante” y “Hexadecimal”.

Copiar: Guarda los 32bits seleccionados.

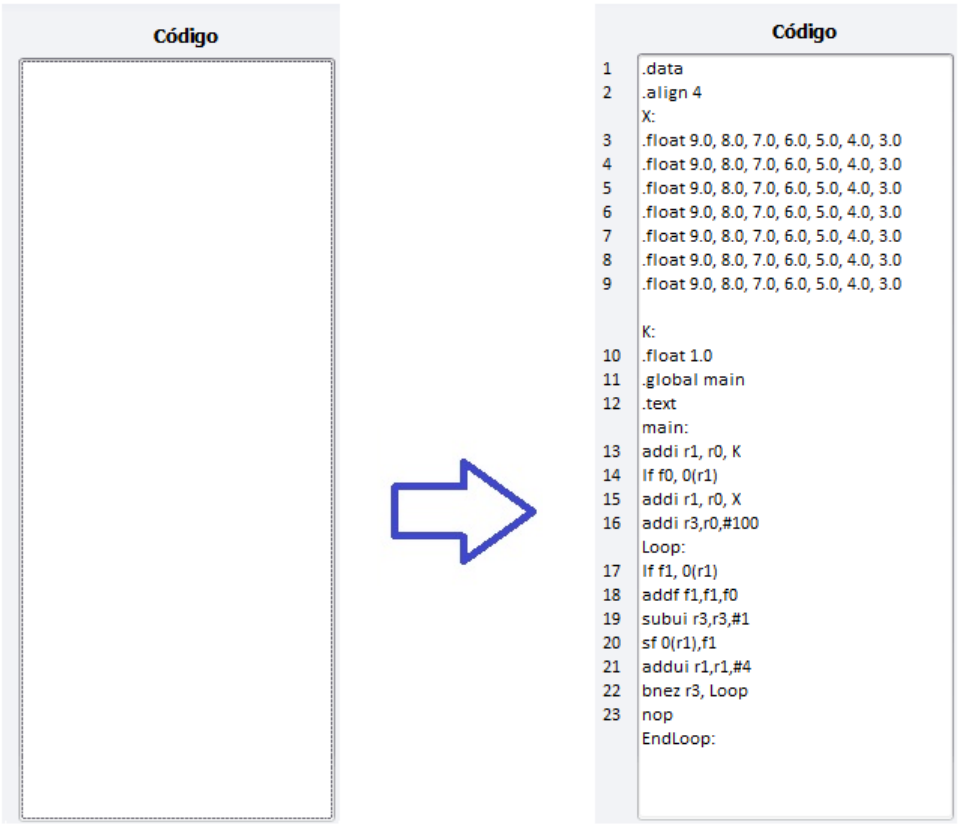
Pegar: Carga en el lugar seleccionado los 32bits que anteriormente copiamos.

Modificar Valor: Abre un nuevo panel en el que se le pide al usuario que introduzca un valor entero para introducírselo a la dirección de memoria seleccionada.

Borrar Valor: Pone los 32bits de la dirección seleccionada a 0.
Limpiar memoria: Borra todas las direcciones de memoria, poniendo a 0 todos los bits.

5.11 Área de Código:

Espacio en blanco donde el usuario debe introducir e código que desea que sea compilado para posteriormente poder ejecutarse.



Mientras el usuario va introduciendo el código, a la izquierda se van contando las líneas de código, exceptuando las líneas con etiquetas y las líneas que solo tienen

comentarios. De esta manera, el usuario puede pulsar con el ratón sobre cualquiera de estos números para generar un punto de interrupción.
Si cargamos un código guardado en un archivo con extensión ".s" también se cargara en ese campo.

5.12 Accesos directos

Los accesos directos representan las mismas funcionalidades que las opciones del menú Ejecutar.

 Nuevo

Borra el área de código y limpia el pipeline y el gráfico.

 Abrir

Abre un código guardado en un archivo con extensión ".s", y lo muestra en el área de código.

 Guardar

Guardar un archivo con un nombre dado por el usuario, con extensión ".s" en la dirección dada.

 Dar Paso (acceso directo F5)

Muestra el resultado del siguiente ciclo de la ejecución, tanto en el pipeline como el gráfico.

 Ejecutar hasta BreakPoint (acceso directo F6)

Si el usuario ha introducido algún punto de interrupción en el código, este se ejecuta hasta ese punto. En caso contrario, el código se ejecutará hasta el final.

 Eliminar BreakPoints

Elimina todos los puntos de interrupción (BreakPoints) que el usuario puso en el código.

 Ejecutar (acceso directo F7)

Muestra el resultado de la ejecución de todos los ciclos que dura el programa que hemos compilado, tanto en el pipeline como el gráfico. Durante esta ejecución se mostrara un nuevo panel en el que el usuario tiene la posibilidad de parar la ejecución en cualquier momento.
Advertencia: No se puede ejecutar un código si anteriormente no se ha compilado.

 Compilar (acceso directo F9)

Se comprueba que el código introducido por el usuario no tiene errores, en caso de que hubiera alguno saldrá un aviso por pantalla. El código queda listo para su ejecución.

 Estadísticas

Muestra en un nuevo panel las estadísticas del código que se está ejecutando.

 Configuración

Abre el panel donde se puede configurar la herramienta DASIT.

 Ayuda

Abre el panel “Acerca de”, donde se muestra el nombre, la versión y los créditos del programa.

Capítulo 6

Caso Ejemplo

A continuación presentamos dos ejemplos de uso para el simulador DLX. Se ha realizado el mismo programa empleando el repertorio de instrucciones DLX en el primer ejemplo y el repertorio multimedia combinado con el repertorio DLX en el segundo ejemplo. El objetivo de dicha comparación es demostrar la mejora que supone la introducción de un conjunto de instrucciones multimedia que operan con varios datos a la vez a parte de mostrar un ejemplo del funcionamiento de la herramienta.

Código

```
1   .data
2   .align 4
    X:
3   .float 9.0, 8.0, 7.0, 6.0, 5.0, 4.0, 3.0, 2.0, 1.0, 0.0
4   .float 9.0, 8.0, 7.0, 6.0, 5.0, 4.0, 3.0, 2.0, 1.0, 0.0
5   .float 9.0, 8.0, 7.0, 6.0, 5.0, 4.0, 3.0, 2.0, 1.0, 0.0
6   .float 9.0, 8.0, 7.0, 6.0, 5.0, 4.0, 3.0, 2.0, 1.0, 0.0
7   .float 9.0, 8.0, 7.0, 6.0, 5.0, 4.0, 3.0, 2.0, 1.0, 0.0
8   .float 9.0, 8.0, 7.0, 6.0, 5.0, 4.0, 3.0, 2.0, 1.0, 0.0
9   .float 9.0, 8.0, 7.0, 6.0, 5.0, 4.0, 3.0, 2.0, 1.0, 0.0
10  .float 9.0, 8.0, 7.0, 6.0, 5.0, 4.0, 3.0, 2.0, 1.0, 0.0
11  .float 9.0, 8.0, 7.0, 6.0, 5.0, 4.0, 3.0, 2.0, 1.0, 0.0
12  .float 9.0, 8.0, 7.0, 6.0, 5.0, 4.0, 3.0, 2.0, 1.0, 0.0
    K:
13  .float 1.0
14  .global main
15  .text
    main:
16  addi r1, r0, K
17  lf f0, 0(r1)
18  addi r1, r0, X
19  addi r3,r0,#100
    Loop:
20  lf f1, 0(r1)
21  addf f1,f1,f0
22  subui r3,r3,#1
23  sf 0(r1),f1
24  addui r1,r1,#4
25  bnez r3, Loop
26  nop
    EndLoop:
```

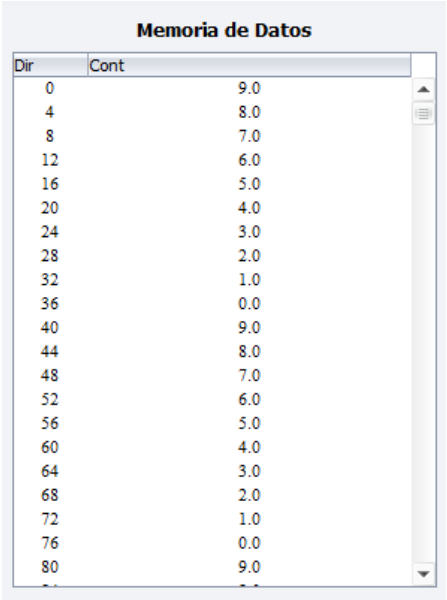
Ejemplo 1: Instrucciones DLX

Código

```
1   .data
2   .align 4
    X:
3   .float 9.0, 8.0, 7.0, 6.0, 5.0, 4.0, 3.0, 2.0, 1.0, 0.0
4   .float 9.0, 8.0, 7.0, 6.0, 5.0, 4.0, 3.0, 2.0, 1.0, 0.0
5   .float 9.0, 8.0, 7.0, 6.0, 5.0, 4.0, 3.0, 2.0, 1.0, 0.0
6   .float 9.0, 8.0, 7.0, 6.0, 5.0, 4.0, 3.0, 2.0, 1.0, 0.0
7   .float 9.0, 8.0, 7.0, 6.0, 5.0, 4.0, 3.0, 2.0, 1.0, 0.0
8   .float 9.0, 8.0, 7.0, 6.0, 5.0, 4.0, 3.0, 2.0, 1.0, 0.0
9   .float 9.0, 8.0, 7.0, 6.0, 5.0, 4.0, 3.0, 2.0, 1.0, 0.0
10  .float 9.0, 8.0, 7.0, 6.0, 5.0, 4.0, 3.0, 2.0, 1.0, 0.0
11  .float 9.0, 8.0, 7.0, 6.0, 5.0, 4.0, 3.0, 2.0, 1.0, 0.0
12  .float 9.0, 8.0, 7.0, 6.0, 5.0, 4.0, 3.0, 2.0, 1.0, 0.0
    K:
13  .float 1.0
14  .global main
15  .text
    main:
16  addi r1, r0, K
17  lvx  m1 r1 m0
18  addi r1, r0, X
19  addi r3,r0,#25
    Loop:
20  lvx  m2 r1 m0
21  vaddfp m2,m2,m1
22  subui r3,r3,#1
23  stvx  m2 r1 m0
24  addui r1,r1,#16
25  bnez r3, Loop
26  nop
    EndLoop:
```

Ejemplo 2: Instrucciones DLX + AltiVec

Al compilar el código, se carga en memoria la secuencia de flotantes 9.0, 8.0, 7.0, 6.0, 5.0, 4.0 3.0, 2.0, 1.0, 0.0, diez veces, ocupando las 400 primeras posiciones de memoria. Esto ocurre de la misma manera en los dos ejemplos propuestos.

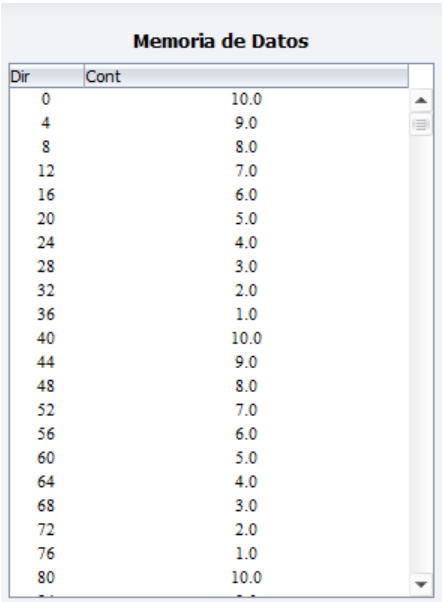


Memoria de Datos

Dir	Cont
0	9.0
4	8.0
8	7.0
12	6.0
16	5.0
20	4.0
24	3.0
28	2.0
32	1.0
36	0.0
40	9.0
44	8.0
48	7.0
52	6.0
56	5.0
60	4.0
64	3.0
68	2.0
72	1.0
76	0.0
80	9.0

El objetivo de ambos ejemplos, es incrementar cada elemento en uno, iterando en un bucle. En el caso simple del DLX solo se incrementa un dato con una instrucción por vuelta del bucle, mientras que el caso que emplea multimedia, permite 4 incrementos por vuelta, con una sola instrucción.

El resultado de la ejecución en ambos casos sería el siguiente.

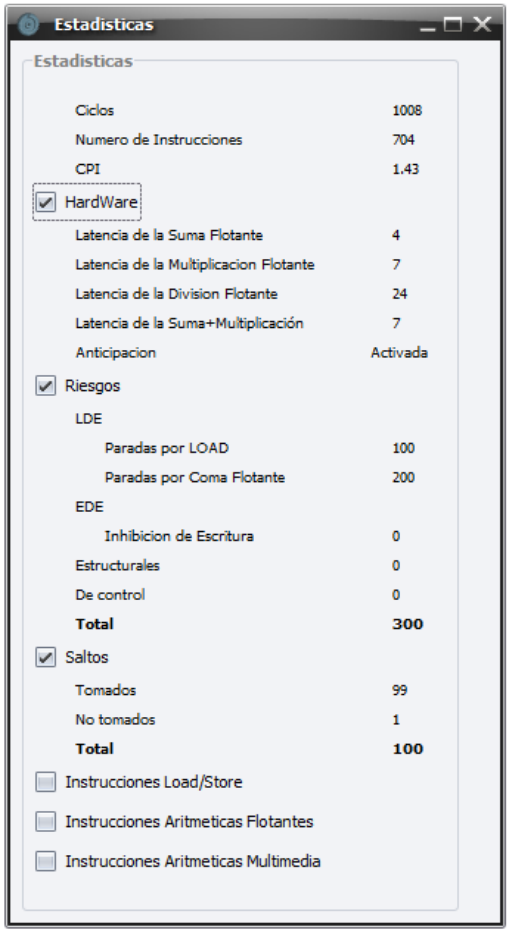


Memoria de Datos

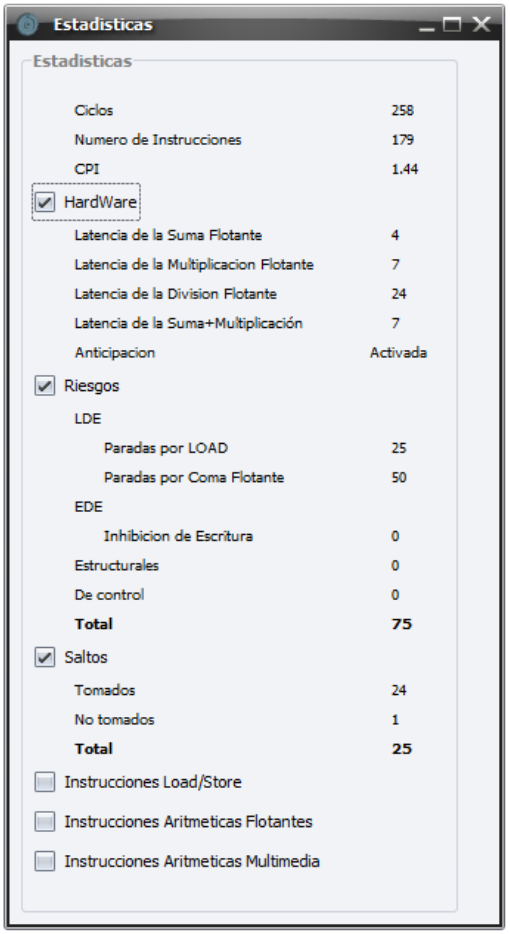
Dir	Cont
0	10.0
4	9.0
8	8.0
12	7.0
16	6.0
20	5.0
24	4.0
28	3.0
32	2.0
36	1.0
40	10.0
44	9.0
48	8.0
52	7.0
56	6.0
60	5.0
64	4.0
68	3.0
72	2.0
76	1.0
80	10.0

Como se puede apreciar gracias a las estadísticas obtenidas, el primer programa tarda en torno a 4 veces más en terminar su ejecución, esto se debe a que la incorporación de instrucciones multimedia que permiten operar con 128 bits en lugar de 32 bits. En este caso en concreto, el uso de las instrucciones, “vaddfp m2, m2, m1” y “stvx m2, r1, m0” permiten en una sola instrucción sumar 4 elementos simultáneamente y almacenarlos en memoria simultáneamente respectivamente.

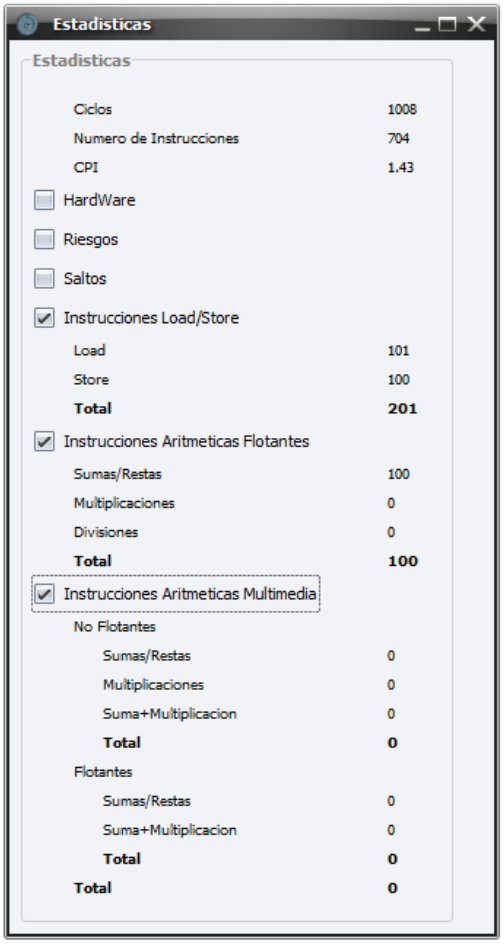
Ejemplo 1: Instrucciones DLX



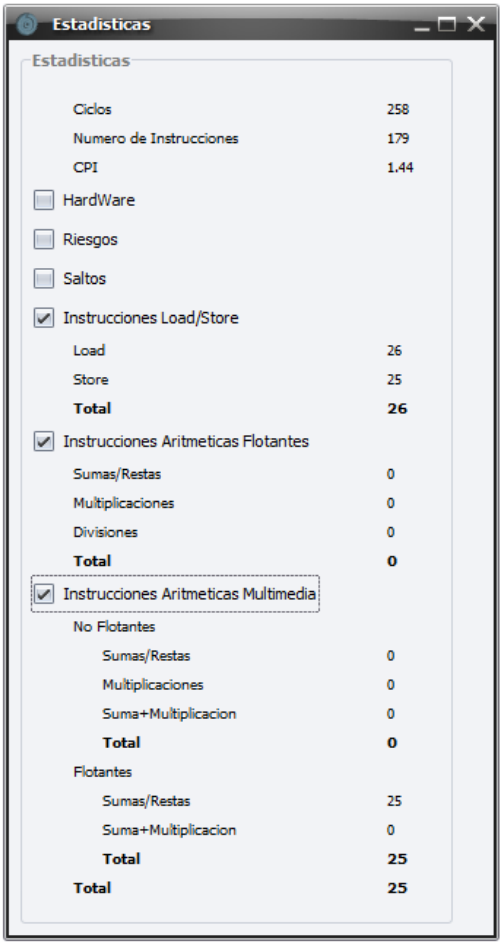
Ejemplo 2: Instrucciones DLX + AltiVec



Ejemplo 1: Instrucciones DLX



Ejemplo 2: Instrucciones DLX + AltiVec



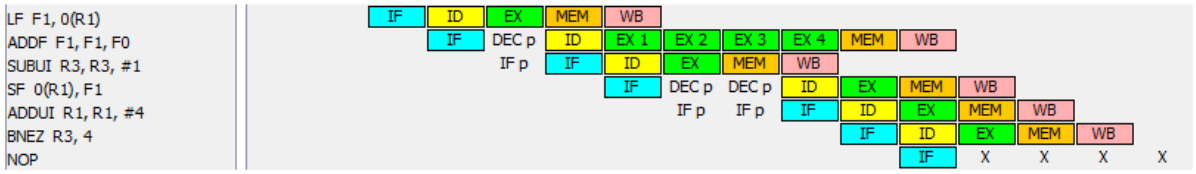
El resto del programa implica en ambos casos los mismos riesgos, pero que se dan 4 veces menos en el segundo ejemplo. Como apreciamos en las imágenes de la ejecución del pipeline, existen paradas por dependencia de datos LDE entre las instrucciones 21 que depende de la 20 y la 23 de la 21.

Igualmente en ambos casos apreciamos que las instrucciones posteriores al salto siempre que este se tome han sido invalidadas (aunque se traten de “nop”s se visualizan como instrucciones anuladas). Lo que se muestra en las estadísticas lo podemos comprobar en el pipeline.

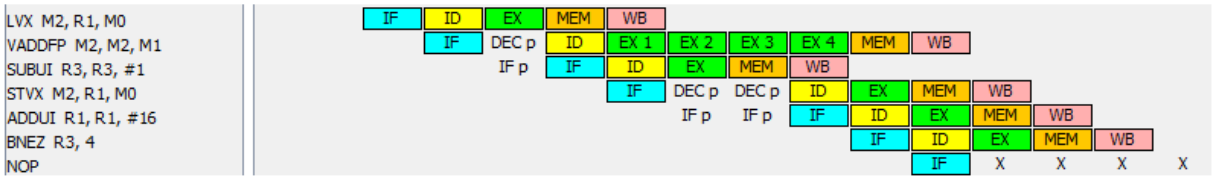
Ambas situaciones se aprecian en las imágenes siguientes.

Estado del pipeline tras una iteración del bucle:

Ejemplo 1: Instrucciones DLX



Ejemplo 2: Instrucciones DLX + AltiVec



Capítulo 7

Conclusiones

Se ha realizado un simulador de un procesador DLX, que incorpora a su repertorio un conjunto de instrucciones multimedia, mediante la máquina de java. Al haber desarrollado el proyecto en java se ha podido obtener un simulador multiplataforma, totalmente funcional en sistemas de Windows, Apple y Linux. El único requisito requerido es tener instalado java. El simulador ha sido implementado por completo sin ninguna librería externa, aparte de las proporcionadas por java, y una librería de "Synthetica", para cambiar la apariencia de la aplicación. La librería utilizada es "syntheticaBlackMoon", para la cual tuvimos que solicitar al servicio de Synthetica, una licencia no comercial. Ademas de eso, no se ha empleado ninguna reutilización de código ya existente.
La herramienta desarrollada permite un fácil uso, debido a que su interfaz resulta sencilla para usuarios que conozcan, aunque sea solo por encima, el comportamiento de un procesador DLX.

La principal aportación de este proyecto es la inclusión del repertorio de instrucciones multimedia AltiVec, totalmente innovadora respecto a otros simuladores existentes, y por ahora única como herramienta didáctica ya que permite estudiar el comportamiento de dicho repertorio sobre un procesador DLX.
En comparación con otras herramientas didácticas similares, nuestro proyecto presenta una innovación en cuanto a la visualización de la ruta de datos a lo largo de la ejecución de las instrucciones. Partiendo de un diagrama explicativo sobre la segmentación de la ruta de datos en un procesador DLX, se puede observar y comprender mejor el comportamiento mediante dicha visualización. No olvidemos mencionar la posibilidad de observar en detalle la ruta de datos en cada etapa de ejecución de las unidades funcionales.

La herramienta final permite al usuario compilar su propio código, siempre que cumpla la especificación dada. Así como depurar su ejecución paso por paso o hasta una instrucción marcada con un punto de interrupción.

Al igual que se incorporan mejoras en la depuración, se tiene la posibilidad de modificar la latencia de las unidades funcionales de la ejecución, que impliquen operaciones en coma flotante.

En todo momento si se desea se pueden modificar los valores de la memoria de datos y del banco de registros manualmente, mediante un sencillo menú. En relación a la memoria y al banco, también se puede modificar el formato de visualización de los datos, convirtiéndolos a hexadecimal, flotantes, enteros, binarios y separándolos por palabras, halfwords o bytes.

Si el usuario tuviese cualquier duda, siempre podría consultar el manual de usuario y el conjunto de instrucciones disponibles desde la ayuda del simulador.
Para aumentar la eficiencia docente de la herramienta se incorporó un conjunto de estadísticas, sobre el comportamiento del pipeline, permitiendo pues un amplio estudio sobre los saltos, burbujas, dependencias y demás para concluir sobre la eficiencia de un programa o para realizar comparación entre diferentes programas.

La herramienta se encuentra a disposición de cualquiera en el enlace siguiente:
http://www.dasit.es.tl/

Palabras Clave

- DLX
- ALTIVEC
- Procesador
- Simulador
- Arquitectura
- Procesador
- Multimedia
- Computador
- Repertorio

Bibliografía

Arquitectura y Micro-Arquitectura

[11] R. Balasubramonian, “Cluster prefetch: tolerating on-chip wire delays in clustered microarchitectures”. Proceedings of the 18th annual international conference on Supercomputing. Malo, France, 2004, pp. 326 - 335

[3] M. Bohr, K. Mistry. Intel's Revolutionary 22nm Transistor Technology. Rob Willoner at Innovation, Junio 2011.

[9] K. Constantinides, O. Mutlu, T. Austin, V. Bertacco, Software-Based Online Detection of Hardware Defects Mechanisms, Architectural Support, and Evaluation. Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture, 2007

[17] K. Diendorff and P.K. Dubey. “How multimedia workloads will change processor design”. IEEE Computer Volume 30, No. 9 (September 1997), 43-45.

[2] M. J. Flynn, P. Hung. “Microprocessor Design Issues: Thoughts on the Road Ahead,” IEEE Micro, vol. 25, no. 3, pp. 16-31, May/June, 2005.

[5] P. Glosekotter, U. Greveler, G. Wirth, “Device degradation and resilient computing”. IEEE International Symposium on Circuits and Systems, 2008. ISCAS 2008.

[1] J.L. Hennesy, D.A. Patterson. “Computer Architecture. A Quantitative Approach”. Morgan Kaufmann Publishers, third edition, 2002.

[26] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, P. Roussel. “The Microarchitecture of the Pentium® 4 Processor”. Intel Technology Journal Q1, 2001.

[8] A. B. Kahng, S. Kang, R. Kumar, J. Sartori, Designing a Processor From the Ground Up to. Allow Voltage/Reliability Tradeoffs. Proceedings del High Performance Computing Architecture 2010.

[16] C. Lee, M. Potkonjak, W. H. Mangione-Smith. “MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communication Systems”. In Proceedings of the Thirtieth Annual IEEE/ACM International Symposium on Microarchitecure (Micro 30). December 1997, pp. 330-335.

[10] Diana Marculescu, Anoop Iyer. “Power efficiency of voltage scaling in multiple clock, multiple voltage cores,” ICCAD, pp. 379-386, 2002 International Conference on Computer-Aided Design

(ICCAD '02), 2002.

[McBh04] C. McNairy, R. Bhatia, “Montecito – The next product in the Itanium® Processor Family”. HotChips 16, 24 August 2004.

Extensiones Multimedia

[24]K. Diefendorff, P. Dubey , R. Hochsprung, H. Scales “AltiVec Extension to PowerPC Accelerates Media Processing”.IEEE Micro, Vol 20(2), Mar/Apr. 2000, pp. 85-96,

[29] T. R. Halfhill - Intel’s Larrabee Redefines GPUs. Septiembre 2008

[18] Intel Corporation. “i860 64-bit Microprocessor”. Data Sheet, Santa Clara, California, Feb. 1989.

[27] Spring03 IDF details Prescott, Microprocessor Report, Marzo 2003

[28] Introducing the 45nm Next-Generation Intel® Core™ Microarchitecture. White paper. Intel 2007

[20] R.B. Lee, “Accelerating Multimedia with Enhanced Microprocessors”. IEEE Micro, Vol 15(2), Mar./Apr. 1995, pp. 22-32.

[22] R.B. Lee, “Subword Parallelism with MAX-2”. IEEE Micro, Vol. 16(4), Jul./Aug. 1996, pp. 51-59.

[19] Motorola. “MC88110 Second Generation RISC Microprocessor. User's Manual”. Austin, Tex, 1991.

[23] S. Oberman, G. Favor and F. Weber “AMD 3Dnow! Technology: Architecture and Implementations”. IEEE Micro, Vol 19(2), Mar./Apr. 1999, pp. 37-58.

[21] M. Tremblay, J. M. O’Connor, V. Narayanan, and L. He, “VIS speeds new media processing”. IEEE Micro, Vol. 16(4), Jul./Aug. 1996, pp. 10–20.

[25] S. Thakkar, T. Huff. “The Internet Streaming SIMD Extensions”. Intel Technology Journal, Q2, 1999.

http://www.freescale.com/files/32bit/doc/ref_manual/ALTIVECPIM.pdf

http://www.freescale.com/webapp/sps/site/overview.jsp?code=DRPPCALTVC

Tecnología

[7] A. Agarwal, C. H. Kim, S. Mukhopadhyay, K. Roy, “Leakage in nano-scale technologies: mechanisms, impact and design considerations”. Proceedings of the 41st annual conference on Design automation. San Diego, CA, USA, 2004, pp: 6 -11.

[4] M. J. Bass, C. M. Christensen. “ The Future of Microprocessor Business”. IEEE Spectrum Online. Abril 2002.

[6] International Technology Roadmap for Semiconductors, “International Technology Roadmap for Semiconductors: 2004 Update”, disponible en http://public.itrs.net/, 2004.

Traducción Binaria

[14] K. Ebcioglu, E. Altman, M. Gschwind, S. Sathaye, “Dynamic Binary Translation and Optimization,” IEEE Transactions on Computers, Vol. 50, No. 6, pp. 529-548, Jun. 2001.

[12] L. Gwennap, “Intel's P6 Uses Decoupled Superscalar Design,” Microprocessor Report, Feb. 16, 1995.

[15] H. Kim, J. E. Smith. “Dynamic Binary Translation for Accumulator-Oriented Architectures,” Proceedings of the International Symposium on Code Generation and Optimization (CGO'03), March 2003, pp. 25 – 35.

[13] R.L. Sites, A. Chernoff, M.B. Kirk, M.P. Marks, and S.G. Robinson. “Binary translation”. Communications of the ACM, 36(2):69-81, February 1993.

Procesador DLX

[36] J. V. Campenhout, P. Verplaetse, H. Neefs. Environment for the Simulation of Computer Architectures for the Purpose of Education. WCAE '98 Proceedings of the 1998 workshop on Computer architecture education. http://trappist.elis.ugent.be/escape/

[37] R.J. Figueiredo, J. A. B. Fortes, R. Eigenmann, N. Kapadia, S. Adabala, J.M.-Alonso, V. Taylor, M. Livny, L. Vidal, J. Chen.A network-computing infrastructure for tool experimentation applied to computer architecture education. WCAE '00, Procs. 2000 Computer architecture education

[34] H. Grunbacher, H. Khosravipour. WinDLX and MIPSim pipeline simulators for teaching computer architecture. Procs. IEEE. Engineering of Computer-Based Systems,1996.

[33] J. L Hennessy,. D. A. Patterson. Computer Architecture: A Quantitative Approach. Morgan Kaufmann, 1996.

[32] D. A. Patterson, J. L. Hennessy. Estructura y diseño de computadores. Reverte, 2000

[38] Miloš Bečvář , Stanislav Kahánek. “VLIW-DLX simulator for educational purposes”, Procs. WCAE '07 Proceedings of the 2007 workshop on Computer architecture education

[35] DLX Simulator (GNU GPL). http://www.davidviner.com/dlx.php

Otras

[30] Resolución 10804. Plan de Estudios de la titulación de Ingeniero en Informática de la UCM. (B.OE. 19 de Mayo de 1997).

[31] Resolución 12977. Acuerdo del Consejo de Universidades (B.O.E. 4 de agosto de 2009).

[39] Philip M. Sailer, David R. Kaeli. “The DLX Instruction Set Architecture Handbook”, 1996. ISBN 1-55860-371-9.

David A. Patterson, John L. Hennessy. “Estuctura y Diseño de Computadores” (Julio 2004) ISBN 84-291-2619-8

Apéndice I

Repertorio de Instrucciones DLX

Índice:

Para todas las instrucciones enteras, se consideran valores de 32 bits en formato de complemento a 2 y sus registros serán tipo R. Por lo tanto el rango de los elementos será R[-2^{31} ; $2^{31}-1$].
Las instrucciones flotantes estarán formadas por el estándar IEEE 754 tanto en simple precisión como en doble, si nos referimos a operandos dobles en cualquiera de las instrucciones esto significara que los operandos los forman registros de tipo F pares dado que un operando doble lo compone 64 bits, lo que equivale a un registro F par y su sucesor.
Los valores de los registros pares flotantes son concatenados respectivamente con los valores de los registros impares, posteriores a ellos, formando operandos de 64 bits. Si nos referimos a flotantes serán operandos flotante de simple precisión lo forman 32 bits, el contenido de un registro flotante.

Indicaciones:

- Rs, Rs1, Rs2, Rs3 se refiere a los registros fuente de una instrucción (para cada instrucción se especificara su tipo, entero o flotante).

- Rd identifica el registro fuente de la operación.

Instrucciones Tipo R

ADD (Integer Add Signed)

Se suma aritméticamente el contenido de los registros fuente enteros Rs1 y Rs2, guardando el resultado en el registro destino entero Rd.

Formato: ADD Rd, Rs1, Rs2

Ejemplo: ADD R3, R7, R4

Operación: rd $<-_{32}$ (rs1) + (rs2)

ADDD (Double-Precision Floating-Point Add Signed)

Los valores de los registros flotantes, han de ser pares, son interpretados como valores dobles y sumados usando una aritmética de punto flotante para formar el resultado de 64 bits, que será almacenado en el banco de registros flotante, en las posiciones Rd y Rd+1.

Formato: ADDD Rd, Rs1, Rs2

Ejemplo: ADDD F2, F0, F4

Operación: rd ||rd+1 $<-_{64}$ [(rs1) || (rs1+1)] + [(rs2) || rs2+1)]

ADDF (Simple-Precision Floating-Point Add Signed)

Se suma aritméticamente el contenido de los registros fuente flotantes Rs1 y Rs2, guardando el resultado en el registro destino flotante Rd.

Formato: ADDF Rd, Rs1, Rs2

Ejemplo: ADDF F3, F7, F4

Operación: rd $<-_{32}$ (rs1) + (rs2)

ADDU (Integer Add Unsigned)

Se suma aritméticamente el contenido de los registros fuente enteros Rs1 y Rs2 sin signo, guardando el resultado sin signo en el registro destino entero Rd.

Formato: ADDU Rd, Rs1, Rs2

Ejemplo: ADDU R3, R7, R4

Operación: rd $<-_{32}$ (rs1) + (rs2)

AND (Logical And)

Se realiza la and lógica del contenido de los registros fuente enteros Rs1 y Rs2, guardando el resultado en el registro destino entero Rd.

Formato: AND Rd, Rs1, Rs2

Ejemplo: AND R3, R7, R4

Operación: rd $<-_{32}$ (rs1) & (rs2)

CVTD2F (Convert DPFP To SPFP)

Se convierte el valor doble contenido en el registro fuente flotante rs1 y rs1+1, en un valor flotante y se guarda en el registro destino flotante Rd.

Formato: CVTD2F Rd, Rs1
(rs1 debe ser par)

Ejemplo: CVTD2F F3, F6

Operación: rd $<-_{32}$ SPFP{DPFP[(rs1) || (rs1 + 1)]}

CVTD2I (Convert DPFP To Integer)

Se convierte el valor doble contenido en el registro fuente flotante rs1 y rs1+1, en un valor entero y se guarda en el registro destino entero Rd.

Formato: CVTD2I Rd, Rs1
(rs1 debe ser par)

Ejemplo: CVTD2I R3, F6

Operación: rd $<-_{32}$ FxPI{DPFP[(rs1) || (rs1 + 1)]}

CVTF2D (Convert SPFP To DPFP)

Se convierte el valor del contenido en el registro fuente flotante rs1 en un valor doble y se guarda en el registro destino flotante Rd y Rd+1.

Formato: CVTF2D Rd, Rs1
(rd debe ser par)

Ejemplo: CVTF2D F4, F1

Operación: rd $<-_{32}$ DPFP {SPFP[rs1]}

CVTF2I (Convert SPFP To Integer)

Se convierte el valor del contenido en el registro fuente flotante rs1 en un valor entero y se guarda en el registro destino entero Rd.

Formato: CVTF2I Rd, Rs1

Ejemplo: CVTF2I R4, F1

Operación: rd $<-_{32}$ FxPI {SPFP[rs1]}

CVTI2D (Convert Integer To DPFP)

Se convierte el valor del contenido en el registro fuente entero rs1 en un valor doble y se guarda en el registro destino flotante par Rd.

Formato: CVTI2D Rd, Rs1
(rd debe ser par)

Ejemplo: CVTI2D F4, R1

Operación: rd $<-_{32}$ DPFP {FxPI[rs1]}

CVTI2F (Convert Integer To DPFP)

Se convierte el valor del contenido en el registro fuente entero rs1 en un valor flotante y se guarda en el registro destino flotante par Rd.

Formato: CVTI2D Rd, Rs1

Ejemplo: CVTI2D F4, R1

Operación: rd $<-_{32}$ DPFP {FxPI[rs1]}

DIV (Integer Divide Signed)

Se divide aritméticamente el contenido de los registros fuente enteros Rs1 entre Rs2, guardando el resultado en el registro destino entero Rd. Si Rs2 es igual a 0 se detiene la ejecución.

Formato: DIV Rd, Rs1, Rs2

Ejemplo: DIV R3, R7, R4

Operación: rd $<-_{32}$ (rs1) / (rs2)

DIVD (Double-Precision Floating-Point Divide Signed)

Los valores de los registros flotantes, han de ser pares, son interpretados como valores dobles y se divide rs1 entre rs2 usando aritmética de punto flotante para formar el resultado de 64 bits, que será almacenado en el banco de registros flotante, en las posiciones Rd y Rd+1. Si Rs2||Rs2+1 es igual a 0 se detiene la ejecución.

Formato: DIVD Rd, Rs1, Rs2

Ejemplo: DIVD F2, F0, F4

Operación: rd ||rd+1 $<-_{64}$ [(rs1) || (rs1+1)] / [(rs2) || rs2+1)]

DIVF (Simple-Precision Floating-Point Divide Signed)

Se divide aritméticamente el contenido de los registros fuente flotantes Rs1 enter Rs2, guardando el resultado en el registro destino flotante Rd.

Formato: DIVF Rd, Rs1, Rs2

Ejemplo: DIVF F3, F7, F4

Operación: rd $<-_{32}$ (rs1) / (rs2)

DIVU (Integer Divide Unsigned)

Se divide aritméticamente el contenido de los registros fuente enteros Rs1 entre Rs2 sin signo, guardando el resultado sin signo en el registro destino entero Rd.

Formato: DIVU Rd, Rs1, Rs2

Ejemplo: DIVU R3, R7, R4

Operación: rd $<-_{32}$ (rs1) / (rs2)

EQD (Set On Equal To DPFP)

Se realiza la comparación entre los valores dobles de los contenidos de los registros flotantes Rs1 y Rs2, si son iguales se escribe un 1 en el registro especial flotante FPSR sino se escribe un 0.

Formato: EQD Rs1, Rs2

Ejemplo: EQD F2, F0

Operación: if [(rs1) || (rs1+1)] = [(rs2) || (rs2+1)] then FPSR <- 1
else FPSR <- 0

EQF (Set On Equal To SPFP)

Se realiza la comparación entre los valores flotantes de los contenidos de los registros flotantes Rs1 y Rs2, si son iguales se escribe un 1 en el registro especial flotante FPSR sino se escribe un 0.

Formato: EQF Rs1, Rs2

Ejemplo: EQF F2, F0

Operación: if (rs1) = (rs2) then FPSR <- 1
else FPSR <- 0

GED (Set On Greater Than Or Equal To DPFP)

Se realiza la comparación entre los valores dobles de los contenidos de los registros flotantes Rs1 y Rs2, si rs1 es mayor o igual a rs2 se escribe un 1 en el registro especial flotante FPSR sino se escribe un 0.

Formato: GED Rs1, Rs2

Ejemplo: GED F2, F0

Operación: if [(rs1) || (rs1+1)] >= [(rs2) || (rs2+1)] then FPSR <- 1
else FPSR <- 0

GEF (Set On Greater Than Or Equal To SPFP)

Se realiza la comparación entre los valores flotantes de los contenidos de los registros flotantes Rs1 y Rs2, si rs1 es mayor o igual a rs2 se escribe un 1 en el registro especial flotante FPSR sino se escribe un 0.

Formato: GEF Rs1, Rs2

Ejemplo: GEF F2, F0

Operación: if (rs1) >= (rs2) then FPSR <- 1
else FPSR <- 0

GTD (Set On Greater Than DPFP)

Se realiza la comparación entre los valores dobles de los contenidos de los registros flotantes Rs1 y Rs2, si rs1 es mayor a rs2 se escribe un 1 en el registro especial flotante FPSR sino se escribe un 0.

Formato: GTD Rs1, Rs2

Ejemplo: GTD F2, F0

Operación: if [(rs1) || (rs1+1)] > [(rs2) || (rs2+1)] then FPSR <- 1
else FPSR <- 0

GTF (Set On Greater Than SPFP)

Se realiza la comparación entre los valores flotantes de los contenidos de los registros flotantes Rs1 y Rs2, si rs1 es mayor rs2 se escribe un 1 en el registro especial flotante FPSR sino se escribe un 0.

Formato: GTF Rs1, Rs2

Ejemplo: GTF F2, F0

Operación: if (rs1) > (rs2) then FPSR <- 1
else FPSR <- 0

LED (Set On Less Than Or Equal To DPFP)

Se realiza la comparación entre los valores dobles de los contenidos de los registros flotantes Rs1 y Rs2, si rs1 es menor o igual a rs2 se escribe un 1 en el registro especial flotante FPSR sino se escribe un 0.

Formato: LED Rs1, Rs2

Ejemplo: LED F2, F0

Operación: if [(rs1) || (rs1+1)] <= [(rs2) || (rs2+1)] then FPSR <- 1
else FPSR <- 0

LEF (Set On Less Than Or Equal To SPFP)

Se realiza la comparación entre los valores flotantes de los contenidos de los registros flotantes Rs1 y Rs2, si rs1 es menor o igual a rs2 se escribe un 1 en el registro especial flotante FPSR sino se escribe un 0.

Formato: LEF Rs1, Rs2

Ejemplo: LEF F2, F0

Operación: if (rs1) <= (rs2) then FPSR <- 1
else FPSR <- 0

LTD (Set On Less Than DPFP)

Se realiza la comparación entre los valores dobles de los contenidos de los registros flotantes Rs1 y Rs2, si rs1 es menor a rs2 se escribe un 1 en el registro especial flotante FPSR sino se escribe un 0.

Formato: LTD Rs1, Rs2

Ejemplo: LTD F2, F0

Operación: if [(rs1) || (rs1+1)] < [(rs2) || (rs2+1)] then FPSR <- 1
else FPSR <- 0

LTF (Set On Less Than SPFP)

Se realiza la comparación entre los valores flotantes de los contenidos de los registros flotantes Rs1 y Rs2, si rs1 es mayor rs2 se escribe un 1 en el registro especial flotante FPSR sino se escribe un 0.

Formato: LTF Rs1, Rs2

Ejemplo: LTF F2, F0

Operación: if (rs1) < (rs2) then FPSR <- 1
else FPSR <- 0

MOVD (Move Double-Precision Floating-Point)

Los contenidos de los registros flotantes fuente Rs1 y Rs1+1 se copian los registros destino flotantes Rd y Rd+ respectivamente. Rs1 y Rd deben ser pares.

Formato: MOVD Rd, Rs1

Ejemplo: MOVD F2, F0

Operación: rd $<-_{32}$ (rs1)
rd+1 $<-_{32}$ (rs1+1)

MOVF (Move Simple-Precision Floating-Point)

El contenido del registro flotante fuente Rs1 se copia al registro destino flotante Rd.

Formato: MOVF Rd, Rs1

Ejemplo: MOVF F2, F0

Operación: rd $<-_{32}$ (rs1)

MOVFP2I (Move SPFP To Integer)

El contenido del registro flotante fuente Rs1 se copia al registro destino entero Rd.

Formato: MOVF Rd, Rs1

Ejemplo: MOVF R2, F0

Operación: rd $<-_{32}$ (rs1)

MOVI2FP (Move Integer To SPFP)

El contenido del registro entero fuente Rs1 se copia al registro destino flotante Rd.
Formato: MOVF Rd, Rs1

Ejemplo: MOVF F2, R0

Operación: rd $<-_{32}$ (rs1)

MULT (Integer Multiply Signed)

Se multiplica aritméticamente el contenido de los registros fuente enteros Rs1 y Rs2, guardando el resultado en el registro destino entero Rd.
Formato: MULT Rd, Rs1, Rs2

Ejemplo: MULT R3, R7, R4

Operación: rd $<-_{32}$ (rs1) x (rs2)

MULTD (Double-Precision Floating-Point Multiply Signed)

Los valores de los registros flotantes, han de ser pares, son interpretados como valores dobles y multiplicados usando una aritmética de punto flotante para formar el resultado de 64 bits, que será almacenado en el banco de registros flotante, en las posiciones Rd y Rd+1.
Formato: MULTD Rd, Rs1, Rs2

Ejemplo: MULTD F2, F0, F4

Operación: rd ||rd+1 $<-_{64}$ [(rs1) || (rs1+1)] x [(rs2) || rs2+1)]

MULTF (Simple-Precision Floating-Point Multiply Signed)

Se multiplica aritméticamente el contenido de los registros fuente flotantes Rs1 y Rs2, guardando el resultado en el registro destino flotante Rd.
Formato: MULTF Rd, Rs1, Rs2

Ejemplo: MULTF F3, F7, F4

Operación: rd $<-_{32}$ (rs1) x (rs2)

MULTU (Integer Multiply Unsigned)

Se multiplica aritméticamente el contenido de los registros fuente enteros Rs1 y Rs2 sin signo, guardando el resultado sin signo en el registro destino entero Rd.

Formato: MULTU Rd, Rs1, Rs2

Ejemplo: MULTU R3, R7, R4

Operación: rd $<-_{32}$ (rs1) x (rs2)

NED (Set On Equal To DPFP)

Se realiza la comparación entre los valores dobles de los contenidos de los registros flotantes Rs1 y Rs2, si no son iguales se escribe un 1 en el registro especial flotante FPSR sino se escribe un 0.

Formato: NED Rs1, Rs2

Ejemplo: NED F2, F0

Operación: if [(rs1) || (rs1+1)] =/= [(rs2) || (rs2+1)] then FPSR <- 1
else FPSR <- 0

NEF (Set On Equal To SPFP)

Se realiza la comparación entre los valores flotantes de los contenidos de los registros flotantes Rs1 y Rs2, si no son iguales se escribe un 1 en el registro especial flotante FPSR sino se escribe un 0.

Formato: NEF Rs1, Rs2

Ejemplo: NEF F2, F0

Operación: if (rs1) =/= (rs2) then FPSR <- 1
else FPSR <- 0

NOP (No Operation)

No hace ninguna operación.

Formato: NOP

Ejemplo: NOP

OR (Logical Or)

Se realiza la or lógica del contenido de los registros fuente enteros Rs1 y Rs2, guardando el resultado en el registro destino entero Rd.

Formato: OR Rd, Rs1, Rs2

Ejemplo: OR R3, R7, R4

Operación: $rd <-_{32} (rs1) \mid (rs2)$

SEQ (Set On Equal To)

Se realiza la comparación entre los valores enteros contenidos en los registros enteros Rs1 y Rs2, si son iguales se escribe un 1 en el registro destino entero Rd sino se escribe un 0.

Formato: SEQ Rd, Rs1, Rs2

Ejemplo: SEQ R4, R2, R0

Operación: if (rs1) = (rs2) then $rd <-_{32} (‘0’^{31} \mid\mid 1)$
else $rd <-_{32} (‘0’^{32})$

SGE (Set On Greater than Or Equal To)

Se realiza la comparación entre los valores enteros contenidos en los registros enteros Rs1 y Rs2, si Rs1 es mayor o igual a Rs2 se escribe un 1 en el registro destino entero Rd sino se escribe un 0.

Formato: SGE Rd, Rs1, Rs2

Ejemplo: SGE R4, R2, R0

Operación: if (rs1) >= (rs2) then $rd <-_{32} (‘0’^{31} \mid\mid 1)$
else $rd <-_{32} (‘0’^{32})$

SGT (Set On Greater than)

Se realiza la comparación entre los valores enteros contenidos en los registros enteros Rs1 y Rs2, si Rs1 es mayor a Rs2 se escribe un 1 en el registro destino entero Rd sino se escribe un 0.

Formato: SGT Rd, Rs1, Rs2

Ejemplo: SGT R4, R2, R0

Operación: if (rs1) > (rs2) then $rd <-_{32} (‘0’^{31} \mid\mid 1)$
else $rd <-_{32} (‘0’^{32})$

SLE (Set On Less than Or Equal To)

Se realiza la comparación entre los valores enteros contenidos en los registros enteros Rs1 y Rs2, si Rs1 es menor o igual a Rs2 se escribe un 1 en el registro destino entero Rd sino se escribe un 0.

Formato: SLE Rd, Rs1, Rs2

Ejemplo: SLE R4, R2, R0

Operación: if (rs1) <= (rs2) then rd $<-_{32}$ ('0'^{31} || 1)
else rd $<-_{32}$ ('0'^{32})

SLT (Set On Less than)

Se realiza la comparación entre los valores enteros contenidos en los registros enteros Rs1 y Rs2, si Rs1 es menor a Rs2 se escribe un 1 en el registro destino entero Rd sino se escribe un 0.

Formato: SLT Rd, Rs1, Rs2

Ejemplo: SLT R4, R2, R0

Operación: if (rs1) < (rs2) then rd $<-_{32}$ ('0'^{31} || 1)
else rd $<-_{32}$ ('0'^{32})

SNE (Set On Not Equal To)

Se realiza la comparación entre los valores enteros contenidos en los registros enteros Rs1 y Rs2, si no son iguales se escribe un 1 en el registro destino entero Rd sino se escribe un 0.

Formato: SNE Rd, Rs1, Rs2

Ejemplo: SNE R4, R2, R0

Operación: if (rs1) =/= (rs2) then rd $<-_{32}$ ('0'^{31} || 1)
else rd $<-_{32}$ ('0'^{32})

SLL (Shift Left Logical)

Se desplaza a la izquierda el contenido del registro fuente entero Rs1 tantos bits como indiquen los 5 bits menos significativos del registro fuente entero Rs2 introduciendo '0' por cada bit desplazado.

Formato: SLL Rd, Rs1, Rs2

Ejemplo: SLL R4, R2, R0

Operación: rd $<-_{32}$ $(rs1)_{d..31}$ || '0'^{s}
Donde $s=(rs2)_{27..31}$

SRA (Shift Right Arithmetic)

Se desplaza a la derecha el contenido del registro fuente entero Rs1 tantos bits como indiquen los 5 bits menos significativos del registro fuente entero Rs2 introduciendo el bit más significativo de Rs1 por cada bit desplazado.

Formato: SRAI Rd, Rs1, Rs2

Ejemplo: SRAI R4, R2, #645

Operación: $rd <-_{32} [(rs1)_0]^s \,||\, (rs1)_{s..31-s}$
$s = (rs2)_{27..31}$

SRLI (Shift Right Arithmetic Immediate)

Se desplaza a la derecha el contenido del registro fuente entero Rs1 tantos bits como indiquen los 5 bits menos significativos del registro fuente entero Rs2 introduciendo '0' por cada bit desplazado.

Formato: SRLI Rd, Rs1, immediate

Ejemplo: SRLI R4, R2, #645

Operación: $rd <-_{32} '0'^s \,||\, (rs1)_{s..31-s}$
$s = (rs2)_{27..31}$

SUB (Integer Substract Signed)

Se resta aritméticamente el contenido del registro fuente entero Rs2 y al contenido del registro fuente entero Rs1, guardando el resultado en el registro destino entero Rd.

Formato: SUB Rd, Rs1, Rs2

Ejemplo: SUB R3, R7, R4

Operación: $rd <-_{32} (rs1) - (rs2)$

SUBD (Double-Precision Floating-Point Substract Signed)

Los valores de los registros flotantes, han de ser pares, son interpretados como valores dobles y restando Rs2 a Rs1, usando una aritmética de punto flotante para formar el resultado de 64 bits, que será almacenado en el banco de registros flotante, en las posiciones Rd y Rd+1.

Formato: SUBD Rd, Rs1, Rs2

Ejemplo: SUBD F2, F0, F4

Operación: $rd \,||\, rd+1 <-_{64} [(rs1) \,||\, (rs1+1)] - [(rs2) \,||\, rs2+1)]$

SUBF (Simple-Precision Floating-Point Substract Signed)

Se resta aritméticamente el contenido del registro fuente flotante Rs2 y al contenido del registro fuente flotante Rs1, guardando el resultado en el registro destino flotante Rd.

Formato: SUBF Rd, Rs1, Rs2

Ejemplo: SUBF F3, F7, F4

Operación: rd $<-_{32}$ (rs1) - (rs2)

SUBU (Integer Substract Unsigned)

Se resta aritméticamente el contenido del registro fuente entero Rs2 y al contenido del registro fuente entero Rs1, ambos sin signo, y se guarda el resultado en el registro destino entero Rd.

Formato: SUBU Rd, Rs1, Rs2

Ejemplo: SUBU R3, R7, R4

Operación: rd $<-_{32}$ (rs1) - (rs2)

XOR (Logical Xor)

Se realiza la xor lógica del contenido de los registros fuente enteros Rs1 y Rs2, guardando el resultado en el registro destino entero Rd.

Formato: OR Rd, Rs1, Rs2

Ejemplo: OR R3, R7, R4

Operación: rd $<-_{32}$ (rs1) XOR (rs2)

Instrucciones Tipo I

ADDI (Integer Add Immediate Signed)

Se suma aritméticamente el contenido del registro fuente entero Rs1 con el resultado de extender el signo al inmediato, guardando el resultado en el registro destino entero Rd.

Formato: ADDI Rd, Rs1, immediate

Ejemplo: ADDI R5, R2, #-645

Operación: rd $<-_{32}$ (rs1) + [$(\text{immediate}_0)^{16}$ || immediate]
Rango del inmediato: R [-2^{15} ; $2^{15}-1$].

ADDUI (Integer Add Immediate Unsigned)

Se suma aritméticamente el contenido del registro fuente entero Rs1 con el resultado de extender el inmediato con 16 '0', guardando el resultado en el registro destino entero Rd.

Formato: ADDUI Rd, Rs1, immediate

Ejemplo: ADDUI R5, R4, #645

Operación: rd $<-_{32}$ (rs1) + [('0')^{16} || immediate]
Rango del inmediato: R[0 ; 2^{16}-1].

ANDI (Logical And Immediate)

Se realiza la and lógica del contenido del registro fuente entero Rs1 con el resultado de extender el inmediato con 16 '0', guardando el resultado en el registro destino entero Rd.

Formato: ANDI Rd, Rs1, immediate

Ejemplo: ANDI R5, R4, #-645

Operación: rd $<-_{32}$ (rs1) & ['0'^{16} || immediate]
Rango del inmediato: R [-2^{15} ; 2^{15}-1].

BEQZ (Branch On Integer Equal To Zero)

Se realiza la comparación entre el contenido del registro fuente entero Rs1 con el valor 0, si son iguales se salta a la dirección indicada por la etiqueta 'name'.

Formato: BEQZ Rs1, 'name'

Ejemplo:
```
BEQZ  R8, Salto
[…]
Salto:

o

Salto:
[…]
BEQZ  R8, Salto
```

Operación: if (rs1) = 0 then PC $<-_{32}$ {[(PC) + 4] + [(name$_0$)^{16} || name]}
En el simulador:
if (rs1) = 0 then PC$<-_{32}$ (name)

BFPF (Branch On Floating-Point False)

Se realiza la comparación entre el contenido del registro especial flotante FPSR con el valor 0, si son iguales se salta a la dirección indicada por la etiqueta 'name'.

Formato: BFPF 'name'

Ejemplo:
```
BFPF Salto
[…]
Salto:

o

Salto:
[…]
BFPF Salto
```

Operación: if (FPSR) = 0 then PC <- $_{32}$ {[(PC) + 4] + [$(name_0)^{16}$ || name]}

En el simulador:

if (FPSR) = 0 then PC<-$_{32}$ (name)

BFPT (Branch On Floating-Point True)

Se realiza la comparación entre el contenido del registro especial flotante FPSR con el valor 1, si son iguales se salta a la dirección indicada por la etiqueta 'name'.

Formato: BFPT 'name'

Ejemplo:
```
BFPT Salto
[…]
Salto:

o

Salto:
[…]
BFPT Salto
```

Operación: if (FPSR) = 1 then PC <- $_{32}$ {[(PC) + 4] + [$(name_0)^{16}$ || name]}

En el simulador:

if (FPSR) = 1 then PC<-$_{32}$ (name)

BNEZ (Branch On Integer Not Equal To Zero)

Se realiza la comparación entre el contenido del registro fuente entero Rs1 con el valor 0, si son distintos se salta a la dirección indicada por la etiqueta 'name'.

Formato: BNEZ Rs1, 'name'

Ejemplo:
```
BNEZ R8, Salto
[…]
Salto:
```

o

```
Salto:
[…]
BNEZ R8, Salto
```

Operación: if (rs1) =/= 0 then PC <- $_{32}$ {[(PC) + 4] + [$(name_0)^{16}$ || name]}

En el simulador:

if (rs1) =/= 0 then PC<-$_{32}$ (name)

JALR (Jump And Link Register)

Se realiza un salto incondicional a la dirección determinada por la etiqueta 'name' y se almacena al valor del PC en el registro R31.

Formato: JALR rs1

Ejemplo: JALR R7

Operación: R31 <-$_{32}$ [(PC) + 8]

PC<-$_{32}$ (rs1)

JR (Jump Register)

Se realiza un salto incondicional a la dirección determinada por el contenido del registro fuente entero Rs1.

Formato: JR Rs1

Ejemplo: JR R4

Operación: PC<-$_{32}$ (rs1)

LB (Load Byte Signed)

Se accede a la dirección de memoria resultante de la suma entre el offset y el valor del registro fuente entero Rs1. El byte accedido se guarda en el registro destino entero Rd después de extender su signo.

Formato: LB Rd, offset(Rs1)

Ejemplo: LB R4, -95(R2)

Operación: $Rd <-_{32} \{M\{[(offset_0)^{16} || offset] + (rs1)\}_0\}^{24} || M\{[offset_0)^{16} || offset] + (rs1)\}$

LBU (Load Byte Unsigned)

Se accede a la dirección de memoria resultante de la suma entre el offset y el valor del registro fuente entero Rs1. El byte accedido se guarda en el registro destino entero Rd después de extenderlo a 32 bits añadiendo ceros.

Formato: LBU Rd, offset(Rs1)

Ejemplo: LBU R4, 95(R2)

Operación: $Rd <-_{32} \text{‘0’}^{24} || M\{[(offset_0)^{16} || offset] + (rs1)\}$

LD (Load Double-Precision Floating-Point)

Se accede a la dirección de memoria resultante de la suma entre el offset y el valor del registro fuente entero Rs1. El doble accedido se guarda en el registro destino flotante Rd. La dirección accedida debe ser múltiplo de 8 de lo contrario se detendrá la ejecución.

Formato: LD Rd, offset(Rs1)
(Rd debe ser par)

Ejemplo: LD F4, -95(R2)

Operación: $Rd || Rd + 1 <-_{64} M\{[(offset_0)^{16} || offset] + (rs1)\}$

LF (Load Single-Precision Floating-Point)

Se accede a la dirección de memoria resultante de la suma entre el offset y el valor del registro fuente entero Rs1. El flotante accedido se guarda en el registro destino flotante Rd. La dirección accedida debe ser múltiplo de 4 de lo contrario se detendrá la ejecución.

Formato: LF Rd, offset(Rs1)

Ejemplo: LF F4, -95(R2)

Operación: $Rd <-_{32} M\{[(offset_0)^{16} || offset] + (rs1)\}$

LH (Load Halfword Signed)

Se accede a la dirección de memoria resultante de la suma entre el offset y el valor del registro fuente entero Rs1. El Halfword accedido se guarda en el registro destino entero Rd después de extender su signo. La dirección accedida debe ser múltiplo de 2 de lo contrario se detendrá la ejecución.

Formato: LH Rd, offset(Rs1)

Ejemplo: LH R4, -95(R2)

Operación: $Rd <-_{32} \{M\{[(offset_0)^{16} || offset] + (rs1)\}_0\}^{16} || M\{[offset_0)^{16} || offset] + (rs1)\}$

LHI (Load High Immediate)

El inmediato de 16 bits se concatena con 16 ceros y el resultado de la operación se almacena en el registro destino entero Rd.

Formato: LHI Rd, immediate

Ejemplo: LHI R4, 595

Operación: $Rd <-_{32}$ immediate || $'0'^{16}$

LHU (Load Halfword Unsigned)

Se accede a la dirección de memoria resultante de la suma entre el offset y el valor del registro fuente entero Rs1. El Halfword accedido se guarda en el registro destino entero Rd después de concatenarlo con 16 ceros. La dirección accedida debe ser múltiplo de 2 de lo contrario se detendrá la ejecución.

Formato: LHU Rd, offset(Rs1)

Ejemplo: LHU R4, -95(R2)

Operación: $Rd <-_{32}$ $'0'^{16}$ || $M\{[offset_0)^{16} || offset] + (rs1)\}$

LW (Load Word)

Se accede a la dirección de memoria resultante de la suma entre el offset y el valor del registro fuente entero Rs1. El dato accedido se guarda en el registro destino entero Rd. La dirección accedida debe ser múltiplo de 4 de lo contrario se detendrá la ejecución.

Formato: LW Rd, offset(Rs1)

Ejemplo: LW R4, -95(R2)

Operación: $Rd <-_{32} M\{[(offset_0)^{16} || offset] + (rs1)\}$

ORI (Logical Or Immediate)

Se realiza la or lógica del contenido del registro fuente entero Rs1 con el resultado de extender el inmediato con 16 '0', guardando el resultado en el registro destino entero Rd.

Formato: ORI Rd, Rs1, immediate

Ejemplo: ORI R5, R4, -645

Operación: rd <-$_{32}$ (rs1) | ['0'^{16} || immediate]
Rango del inmediato: R [-2^{15} ; 2^{15}-1].

SB (Store Byte)

El byte menos significativo del registro destino entero Rd se guarda en la dirección de memoria resultante de la suma entre el offset y el valor del registro fuente entero Rs1.

Formato: SB offset(Rs1), Rd

Ejemplo: SB -95(R2), R3

Operación: M{[offset$_0$)^{16} || offset] + (rs1)} <-$_8$ (Rd)$_{24..31}$

SD (Store Double-Precision Floating-Point)

El doble formado por los registros destino flotantes Rd y Rd+1 se almacena en la dirección de memoria resultante de la suma entre el offset y el valor del registro fuente entero Rs1. La dirección accedida debe ser múltiplo de 8 de lo contrario se detendrá la ejecución.

Formato: SD offset(Rs1), Rd
(Rd debe ser par)

Ejemplo: SD -95(R2), F4

Operación: M{[(offset$_0$)^{16} || offset] + (rs1)} <-$_{64}$ (Rd) || (Rd + 1)

SF (Store Single-Precision Floating-Point)

El contenido del registro destino flotante Rd se guarda en la dirección de memoria resultante de la suma entre el offset y el valor del registro fuente entero Rs1. La dirección accedida debe ser múltiplo de 4 de lo contrario se detendrá la ejecución.

Formato: SF offset(Rs1), Rd

Ejemplo: SF -95(R2), F4

Operación: M{[(offset$_0$)^{16} || offset] + (rs1)} <-$_{32}$ (Rd)

SH (Store Halfword)

El HalfWord menos significativo del registro destino entero Rd se guarda en la dirección de memoria resultante de la suma entre el offset y el valor del registro fuente entero Rs1. La dirección accedida debe ser múltiplo de 2 de lo contrario se detendrá la ejecución.

Formato: SH offset(Rs1), Rd

Ejemplo: SH -95(R2), R5

Operación: $M\{[offset_0)^{16} \,||\, offset] + (rs1)\} <-_{16} (Rd)_{16..31}$

SW (Load Word)

Se accede a la dirección de memoria resultante de la suma entre el offset y el valor del registro fuente entero Rs1. El dato accedido se guarda en el registro destino entero Rd. La dirección accedida debe ser múltiplo de 4 de lo contrario se detendrá la ejecución.

Formato: SW offset(Rs1), Rd

Ejemplo: SW -95(R2), R4

Operación: $M\{[(offset_0)^{16} \,||\, offset] + (rs1)\} <-_{32} (Rd)$

SEQI (Set On Equal To Immediate)

Se realiza la comparación entre el valor entero contenido en el registro entero Rs1 y el inmediato, si son iguales se escribe un 1 en el registro destino entero Rd sino se escribe un 0.

Formato: SEQI Rd, Rs1, immediate

Ejemplo: SEQI R4, R2, #648

Operación: if (rs1) = $(immediate_0)^{16}$ || immediate then rd <-$_{32}$ ('0'^{31} || 1)
else rd <-$_{32}$ ('0'^{32})

SGEI (Set On Greater Than Or Equal To Immediate)

Se realiza la comparación entre el valor entero contenido en el registro entero Rs1 y el inmediato, si Rs1 es mayor o igual al inmediato se escribe un 1 en el registro destino entero Rd sino se escribe un 0.

Formato: SGEI Rd, Rs1, immediate

Ejemplo: SGEI R4, R2, #648

Operación: if (rs1) >= $(immediate_0)^{16}$ || immediate then rd <-$_{32}$ ('0'^{31} || 1)
else rd <-$_{32}$ ('0'^{32})

SGTI (Set On Greater Than Immediate)

Se realiza la comparación entre el valor entero contenido en el registro entero Rs1 y el inmediato, si Rs1 es mayor al inmediato se escribe un 1 en el registro destino entero Rd sino se escribe un 0.

Formato: SGTI Rd, Rs1, immediate

Ejemplo: SGTI R4, R2, #648

Operación: if (rs1) > $(immediate_0)^{16}$ || immediate then rd <-$_{32}$ ('0'^{31} || 1)
else rd <-$_{32}$ ('0'^{32})

SLEI (Set On Less Than Or Equal To Immediate)

Se realiza la comparación entre el valor entero contenido en el registro entero Rs1 y el inmediato, si Rs1 es menor o igual al inmediato se escribe un 1 en el registro destino entero Rd sino se escribe un 0.

Formato: SLEI Rd, Rs1, immediate

Ejemplo: SLEI R4, R2, #648

Operación: if (rs1) <= $(immediate_0)^{16}$ || immediate then rd <-$_{32}$ ('0'^{31} || 1)
else rd <-$_{32}$ ('0'^{32})

SLTI (Set On Less Than Or Equal To Immediate)

Se realiza la comparación entre el valor entero contenido en el registro entero Rs1 y el inmediato, si Rs1 es menor al inmediato se escribe un 1 en el registro destino entero Rd sino se escribe un 0.

Formato: SLEI Rd, Rs1, immediate

Ejemplo: SLEI R4, R2, #648

Operación: if (rs1) < $(immediate_0)^{16}$ || immediate then rd <-$_{32}$ ('0'^{31} || 1)
else rd <-$_{32}$ ('0'^{32})

SNEI (Set On Not Equal To Immediate)

Se realiza la comparación entre el valor entero contenido en el registro entero Rs1 y el inmediato, si no son iguales se escribe un 1 en el registro destino entero Rd sino se escribe un 0.

Formato: SNEI Rd, Rs1, immediate

Ejemplo: SNEI R4, R2, #648

Operación: if (rs1) =/= $(immediate_0)^{16}$ || immediate then rd <-$_{32}$ ('0'^{31} || 1)
else rd <-$_{32}$ ('0'^{32})

SLLI (Shift Left Logical Immediate)

Se desplaza a la izquierda el contenido del registro fuente entero Rs1 tantos bits como indiquen los 5 bits menos significativos del inmediato introduciendo '0' por cada bit desplazado.

Formato: SLLI Rd, Rs1, immediate

Ejemplo: SLLI R4, R2, #645

Operación: $rd <-_{32} (rs1)_{s..31} || '0'^{s}$
$s=(immediate)_{27..31}$

SRAI (Shift Right Arithmetic Immediate)

Se desplaza a la derecha el contenido del registro fuente entero Rs1 tantos bits como indiquen los 5 bits menos significativos del inmediato introduciendo el bit más significativo de Rs1 por cada bit desplazado.

Formato: SRAI Rd, Rs1, immediate

Ejemplo: SRAI R4, R2, #645

Operación: $rd <-_{32} [(rs1)_0]^{s} || (rs1)_{s..31-s}$
$s = (immediate)_{27..31}$

SRLI (Shift Right Arithmetic Immediate)

Se desplaza a la derecha el contenido del registro fuente entero Rs1 tantos bits como indiquen los 5 bits menos significativos del inmediato introduciendo '0' por cada bit desplazado.

Formato: SRLI Rd, Rs1, immediate

Ejemplo: SRLI R4, R2, #645

Operación: $rd <-_{32} '0'^{s} || (rs1)_{s..31-s}$
$s = (immediate)_{27..31}$

SUBI (Integer Substract Immediate Signed)

Se resta aritméticamente el resultado de extender el signo al inmediato al contenido del registro fuente entero Rs1 con, guardando el resultado en el registro destino entero Rd.

Formato: SUBI Rd, Rs1, immediate

Ejemplo: SUBI R5, R2, #-645

Operación: $rd <-_{32} (rs1) + [(immediate_0)^{16} || immediate]$
Rango del inmediato: R $[-2^{15} ; 2^{15}-1]$.

SUBUI (Integer Substract Immediate Unsigned)

Se resta aritméticamente el resultado de extender el inmediato con 16 '0' al contenido del registro fuente entero Rs1 con, guardando el resultado en el registro destino entero Rd.

Formato: SUBUI Rd, Rs1, immediate

Ejemplo: SUBUI R5, R4, #645

Operación: rd <-$_{32}$ (rs1) - [('0')^{16} || immediate]
Rango del inmediato: R [0 ; 2^{16}-1].

XORI (Logical Xor Immediate)

Se realiza la xor lógica del contenido del registro fuente entero Rs1 con el resultado de extender el inmediato con 16 '0', guardando el resultado en el registro destino entero Rd.

Formato: XORI Rd, Rs1, immediate

Ejemplo: XORI R5, R4, -645

Operación: rd <-$_{32}$ (rs1) XOR ['0'^{16} || immediate]
Rango del inmediato: R [-2^{15} ; 2^{15}-1].

Instrucciones Tipo J

J (Jump)

Se realiza un salto incondicional a la dirección determinada por la etiqueta 'name'.

Formato: J 'name'

Ejemplo:
```
J Salto
[…]
Salto:

o

Salto:
[…]
J Salto
```

Operación: PC <- $_{32}$ {[(PC) + 4] + [(name$_0$)^{16} || name]}
En el simulador:
PC<-$_{32}$ (name)

JAL (Jump And Link)

Se realiza un salto incondicional a la dirección determinada por la etiqueta 'name' y se almacena al valor del PC en el registro R31.

Formato: JAL 'name'

Ejemplo:
```
JAL Salto
[…]
Salto:

o

Salto:
[…]
JAL Salto
```

Operación:

$PC <-_{32} \{[(PC) + 4] + [(name_0)^{16} \,||\, name]\}$

En el simulador:

$R31 <-_{32} [(PC) + 8]$

$PC <-_{32} (name)$

RFE (Return From Exception)

Se carga en PC el contenido del registro IAR.

Formato: RFE

Ejemplo: RFE

Operación: $PC <-_{32} (IAR)$

TRAP

Se realiza un salto incondicional a la dirección determinada por la etiqueta 'name'. Guardando el PC +8 en el registro especial IAR para un retorno.

Formato: TRAP 'name'

Ejemplo:
```
TRAP Salto
[…]
Salto:

o

Salto:
[…]
TRAP Salto
```

Operación:

$IAR <-_{32} [(PC) + 8]$

$PC <-_{32} (name)$

Apéndice II

Directivas del DLX

Todos los programas DLX tienen una serie de directivas destinadas a cargar datos en la memoria previo a la ejecución del código. Estas directivas son una serie de palabras reservadas, de las cuales las dos más genéricas son “data” y “text”.

DATA:

La directiva “data” determina donde comienza la región de datos en la cual declararemos los datos que utilizaremos en la ejecución de nuestro programa. Dentro de esta región tenemos diferentes directivas para poder guardar diferentes tipos de datos.

Formato: Data
Ejemplo: .data

TEXT:

La directiva “text” determina donde comienza la región de código donde podremos escribir las instrucciones de nuestro programa.

Formato: Text
Ejemplo: .text

Ademas de estas dos, dentro de la región de datos, podemos encontrar las siguientes:

ALIGN:

Sirve para poder alinear los datos que estamos introduciendo de forma sean cargados en la parte más alta de la dirección con los n bits más bajos puestos a cero.

Formato: Align n
Ejemplo: .align 4

ASCII:

Almacena en memoria cadenas de caracteres.

Formato: Ascii “cadena1”, “cadena2”
Ejemplo: .ascii “hola”,”bien”

ASCIIZ:

Almacena en memoria cadenas de caracteres y al final de la cadena almacena el caracter NULL.

Formato: Asciiz “cadena1”, “cadena2”, …
Ejemplo: .asciiz “hola”,”bien”

BYTE:

Introduce una secuencia de bytes en memoria.

Formato: Byte “byte1”, “byte2”, …
Ejemplo: .byte “10”,”34”

DOUBLE:

Introduce la secuencia de números en memoria como números en coma flotante de precisión doble.

Formato: Double “double1”, “double2”, …
Ejemplo: .double “4.3424”,”8.234”

FLOAT:

Introduce la secuencia de números en memoria como números en coma flotante de precisión simple.

Formato: Float “float1”, “float2”, …
Ejemplo: .float “4.3424”,”8.234”

WORD:

Introduce la secuencia de palabras en memoria.

Formato: Word “b1”, “b2”, …
Ejemplo: .word “525”,”8234”

GLOBAL:

Permite que una etiqueta pueda ser referenciada en cualquier parte de la región de código.

Formato: Global etiqueta
Ejemplo: .global main

SPACE:

Reserva n posiciones de memoria y las deja libres.

Formato: Space n
Ejemplo: .space 13

Apéndice III

Repertorio de Instrucciones ALTIVEC

Para las instrucciones que operan a nivel de byte se considerara dependiendo de la operación byte con signo o byte sin signo. Para las operaciones con byte con signo el rango será R[-2^7 ; 2^7-1] y para byte sin signo el rango será R[0 ; 2^8-1].
Para las instrucciones que operan a nivel de half word se consideraran dependiendo de la operación half word con signo o half word sin signo. Para las operaciones con half word con signo el rango será R[-2^{15};2^{15}-1] y para half word sin signo el rango será R[0;2^{16}-1].
Para las instrucciones que operan a nivel de enteros se consideraran dependiendo de la operación entero con signo o entero sin signo. Para las operaciones con entero con signo el rango será R[-2^{31};2^{31}-1] y para entero sin signo el rango será R[0;2^{32}-1].
Las instrucciones flotantes estarán formadas por el estándar IEEE 754 en precisión simple.

Indicaciones:

- Rs, Rs1, Rs2, Rs3 se refiere a los registros fuente de una instrucción (para cada instrucción se especificara su tipo).

- Rd identifica el registro fuente de la operación.

Instrucciones Tipo R Multimedia

MFVSCR(Vector Move from Vector Status and Control Register)

El registro destino multimedia Rd guarda en sus cuatro bytes más bajos el registro VCSR dejando el resto de elementos a 0.

Formato: MFVSCR Rd

Ejemplo: MFVSCR m7

Operacion: $Rd_{[0-96]} <- 0$
$Rd_{[96-127]} <-_{32} VCSR$

MTVSCR(Vector Move to Vector Status and Control Register)

Se guarda en el registro VSCR los últimos cuatro bytes del registro fuente multimedia Rs1.

Formato: MTVSCR Rs1

Ejemplo: MTVSCR m4

Operacion: $VCSR <-_{32} Rs1_{[96-127]}$

VADDCUW(Vector Add Carryout Unsigned Word)

Se suma cada palabra del registro fuente multimedia de Rs1 con la correspondiente palabra del registro multimedia Rs2. El "carry" de cada suma se extiende con ceros y se guarda en el elemento correspondiente del registro destino multimedia Rd.

Formato: VADDCUW Rd Rs1 Rs2

Ejemplo: VADDCUW m3 m2 m1

Operacion: do i=0 to 3
$Rd_i <-_{32} CarryOut(Rs1_i + Rs2_i)$

VADDFP (Vector Float Add)

Se suma cada elemento flotante del registro fuente multimedia Rs1 con el correspondiente elemento flotante del registro fuente multimedia Rs2, guardando el resultado en el elemento correspondiente del registro destino multimedia Rd.
Si VSCR[NJ] =1 cada operador denormalizado se trunca a cero conservando su signo antes de la operación, si la operación y cada elemento denormalizado del resultado es truncado a cero con el mismo signo.

Formato: VADDFP Rd Rs1 Rs2

Ejemplo: VADDFP m3 m2 m1

Operacion: do i=0 to 3
$Rd_i <-_{32} Rs1_i + Rs2_i$

VADDSBS (Vector Signed Byte Add Saturated)

Se suma cada byte con signo del registro fuente multimedia Rs1 con el correspondiente byte con signo del registro fuente multimedia Rs2, guardando el resultado en el byte correspondiente del registro destino multimedia Rd.
Si hay saturación en la suma, el registro VSCR[SAT] se pone a uno.

Formato: VADDSBS Rd Rs1 Rs2

Ejemplo: VADDSBS m3 m2 m1

Operacion: do i=0 to 16
$Rd_i <-_{8} Saturacion(Rs1_i + Rs2_i)$

VADDSHS (Vector Signed Half Word Saturated)

Se suma cada media palabra con signo del registro fuente multimedia Rs1 con la correspondiente media palabra con signo del registro fuente multimedia Rs2, guardando el resultado en cada media palabra correspondiente del registro destino multimedia Rd.
Si hay saturación en la suma, el registro VSCR[SAT] se pone a uno.

Formato: VADDSHS Rd Rs1 Rs2

Ejemplo: VADDSHS m3 m2 m1

Operacion: do i=0 to 8
$Rd_i <-_{16} Saturacion (Rs1_i + Rs2_i)$

VADDSWS (Vector Signed Word Saturated)

Se suma cada palabra con signo del registro fuente multimedia Rs1 con la correspondiente palabra con signo del registro fuente multimedia Rs2, guardando el resultado en cada palabra correspondiente del registro destino multimedia Rd.
Si hay saturación en la suma, el registro VSCR[SAT] se pone a uno.

Formato: VADDSWS Rd Rs1 Rs2

Ejemplo: VADDSWS m3 m2 m1

Operacion: do i=0 to 4
$Rd_i <-_{32}$ Saturacion ($Rs1_i + Rs2_i$)

VADDUBM(Vector Byte Add)

Se suma cada byte del registro fuente multimedia Rs1 con el correspondiente byte del registro fuente multimedia Rs2, guardando el resultado en el byte correspondiente del registro destino multimedia Rd.

Formato: VADDUBM Rd Rs1 Rs2

Ejemplo: VADDUBM m3 m2 m1

Operacion: do i=0 to 16
$Rd_i <-_{8}$ $Rs1_i + Rs2_i$

VADDUBS(Vector Unsigned Byte Add Saturated)

Se suma cada byte sin signo del registro fuente multimedia Rs1 con el correspondiente byte sin signo del registro fuente multimedia Rs2, guardando el resultado en byte correspondiente del registro destino multimedia Rd.
Si hay saturación en la suma, el registro VSCR[SAT] se pone a uno.

Formato: VADDUBS Rd Rs1 Rs2

Ejemplo: VADDUBS m3 m2 m1

Operacion: do i=0 to 16
$Rd_i <-_{8}$ Saturacion ($Rs1_i + Rs2_i$)

VADDUHM (Vector Half Word Add)

Se suma cada media palabra del registro fuente multimedia Rs1 con la correspondiente media palabra del registro fuente multimedia Rs2, guardando el resultado en la media palabra correspondiente del registro destino multimedia Rd.

Formato: VADDUHM Rd Rs1 Rs2

Ejemplo: VADDUHM m3 m2 m1

Operacion: do i=0 to 8
$Rd_i <-_{16} Rs1_i + Rs2_i$

VADDUHS (Vector Unsigned Half Word Add Saturated)

Se suma cada media palabra sin signo del registro fuente multimedia Rs1 con la correspondiente media palabra sin signo del registro fuente multimedia Rs2, guardando el resultado en cada media palabra correspondiente del registro destino multimedia Rd.
Si hay saturación en la suma, el registro VSCR[SAT] se pone a uno.

Formato: VADDUHS Rd Rs1 Rs2

Ejemplo: VADDUHS m3 m2 m1

Operacion: do i=0 to 8
$Rd_i <-_{16}$ Saturacion $(Rs1_i + Rs2_i)$

VADDUWM (Vector Word Add)

Se suma cada palabra del registro fuente multimedia Rs1 con la correspondiente palabra del registro fuente multimedia Rs2, guardando el resultado en la palabra correspondiente del registro destino multimedia Rd.

Formato: VADDUWM Rd Rs1 Rs2

Ejemplo: VADDUWM m3 m2 m1

Operacion: do i=0 to 4
$Rd_i <-_{32} Rs1_i + Rs2_i$

VADDUWS (Vector Unsigned Word Add Saturated)

Se suma cada palabra con signo del registro fuente multimedia Rs1 con la correspondiente palabra con signo del registro fuente multimedia Rs2, guardando el resultado en cada palabra correspondiente del registro destino multimedia Rd.
Si hay saturación en la suma, el registro VSCR[SAT] se pone a uno.

Formato: VADDUWS Rd Rs1 Rs2

Ejemplo: VADDUWS m3 m2 m1

Operacion: do i=0 to 4
$Rd_i <-_{32}$ Saturacion $(Rs1_i + Rs2_i)$

VAND (Vector Logical AND)

El registro destino multimedia Rd es el resultado de hacer la operación lógica And bit a bit del los registros fuentes multimedia Rs1 y Rs2.

Formato: VAND Rd Rs1 Rs2

Ejemplo: VAND m3 m2 m1

Operacion: $Rd <-_{128}$ Rs1 & Rs2

VANDC (Vector Logical AND whit Complement)

El registro destino multimedia Rd es el resultado de hacer la operación lógica And bit a bit del los registros fuentes multimedia Rs1 y el complemento de Rs2.

Formato: VANDC Rd Rs1 Rs2

Ejemplo: VANDC m3 m2 m1

Operación: $Rd <-_{128}$ Rs1 & ⍰ Rs2

VAVGSB (Vector Signed Byte Average)

Cada byte con signo del registro destino multimedia Rd es el resultado de hacer la media de cada byte con signo correspondiente a los registros fuente multimedia Rs1 y Rs2.

Formato: VAVGSB Rd Rs1 Rs2

Ejemplo: VAVGSB m3 m2 m1

Operación: do i = 0 to 15
$Rd_i <-_8 (Rs1_i + Rs2_i + 1)/2$

VAVGSH(Vector Signed Half Word Average)

Cada media palabra con signo del registro destino multimedia Rd es el resultado de hacer la media de cada media palabra con signo correspondiente a los registros fuente multimedia Rs1 y Rs2.

Formato: VAVGSH Rd Rs1 Rs2

Ejemplo: VAVGSH m3 m2 m1

Operación: do i = 0 to 7
$Rd_i <-_{16} (Rs1_i + Rs2_i + 1)/2$

VAVGSW (Vector Signed Word Average)

Cada palabra con signo del registro destino multimedia Rd es el resultado de hacer la media de cada palabra con signo correspondiente a los registros fuente multimedia Rs1 y Rs2.

Formato: VAVGSW Rd Rs1 Rs2

Ejemplo: VAVGSW m3 m2 m1

Operación: do i = 0 to 3
$Rd_i <-_{32} (Rs1_i + Rs2_i + 1)/2$

VAVGUB (Vector Unsigned Byte Average)

Cada byte sin signo del registro destino multimedia Rd es el resultado de hacer la media de cada byte sin signo correspondiente a los registros fuente multimedia Rs1 y Rs2.

Formato: VAVGUB Rd Rs1 Rs2

Ejemplo: VAVGUB m3 m2 m1

Operación: do i = 0 to 15
$Rd_i <-_{8} (Rs1_i + Rs2_i + 1)/2$

VAVGUH (Vector Unsigned Half Word Average)

Cada media palabra sin signo del registro destino multimedia Rd es el resultado de hacer la media de cada media palabra sin signo correspondiente a los registros fuente multimedia Rs1 y Rs2.

Formato: VAVGUH Rd Rs1 Rs2

Ejemplo: VAVGUH m3 m2 m1

Operación: do i = 0 to 7
$Rd_i <-_{16} (Rs1_i + Rs2_i + 1)/2$

VAVGUW (Vector Unsigned Word Average)

Cada palabra sin signo del registro destino multimedia Rd es el resultado de hacer la media de cada palabra sin signo correspondiente a los registros fuente multimedia Rs1 y Rs2.

Formato: VAVGUW Rd Rs1 Rs2

Ejemplo: VAVGUW m3 m2 m1

Operación: do i = 0 to 3
$Rd_i <-_{32} (Rs1_i + Rs2_i + 1)/2$

VCMPBFP (Vector Compare Bounds Floating-Point)

Cada elemento flotante del registro fuente multimedia Rs1 es comparado con el elemento correspondiente del registro fuente multimedia Rs2. El bit 0 de cada elemento del registro destino multimedia Rd es 0 si el elemento correspondiente del registro Rs1 es menor o igual que el elemento correspondiente del registro Rs2, en otro caso vale 1. El bit 1 de cada elemento del registro destino es 0 si el elemento del registro Rs1 es mayor o igual que el elemento negativo del registro Rs2, en otro caso vale 1. El resto de bits del registro destino se ponen a 0.
Si VSCR[NJ] =1 cada operador denormalizado se trunca a cero antes de la operación.

Formato: VCMPBFP Rd Rs1 Rs2

Ejemplo: VCMPBFP m3 m2 m1

Operación: do i = 0 to 3
$Rd_i <-_{32} 0$
If($Rs1_i \leq Rs2_i$)
then $Rd_i[0]$ <- 0
else $Rd_i[0]$ <- 1
if($Rs1_i \geq -Rs2_i$)
then $Rd_i[1]$ <- 0
else $Rd_i[1]$ <- 1

VCMPEQFP (Vector Float Compare Equal)

Cada elemento flotante del registro fuente multimedia Rs1 es comparado con el elemento flotante correspondiente del registro fuente Rs2, si son iguales el elemento correspondiente del registro destino multimedia Rd se rellena de unos en caso contrario con ceros.
Si VSCR[NJ] =1 cada operador denormalizado se trunca a cero antes de la operación.

Formato: VCMPEQFP Rd Rs1 Rs2

Ejemplo: VCMPEQFP m3 m2 m1

Operación: do i = 0 to 3
If($Rs1_i = Rs2_i$)
then $Rd_i <-_{32} 1$
else $Rd_i <-_{32} 0$

VCMPEQUB (Vector Unsigned Byte Compare Equal)

Cada byte sin signo del registro fuente multimedia Rs1 es comparado con cada byte sin signo correspondiente del registro fuente Rs2, si son iguales el elemento correspondiente del registro destino multimedia Rd se rellena de unos en caso contrario con ceros.

Formato: VCMPEQUB Rd Rs1 Rs2

Ejemplo: VCMPEQUB m3 m2 m1

Operación: do i = 0 to 15
If($Rs1_i = Rs2_i$)
then $Rd_i <-_{8} 1$
else $Rd_i <-_{8} 0$

VCMPEQUH (Vector Unsigned Half Word Compare Equal)

Cada media palabra sin signo del registro fuente multimedia Rs1 es comparado con cada media palabra sin signo correspondiente del registro fuente Rs2, si son iguales el elemento correspondiente del registro destino multimedia Rd se rellena de unos en caso contrario con ceros.

Formato: VCMPEQUH Rd Rs1 Rs2

Ejemplo: VCMPEQUH m3 m2 m1

Operación: do i = 0 to 7
If($Rs1_i = Rs2_i$)
then $Rd_i <-_{16} 1$
else $Rd_i <-_{16} 0$

VCMPEQUW (Vector Unsigned Word Compare Equal)

Cada palabra sin signo del registro fuente multimedia Rs1 es comparado con cada palabra sin signo correspondiente del registro fuente Rs2, si son iguales el elemento correspondiente del registro destino multimedia Rd se rellena de unos en caso contrario con ceros.

Formato: VCMPEQUW Rd Rs1 Rs2

Ejemplo: VCMPEQUW m3 m2 m1

Operación:
do i = 0 to 3
If($Rs1_i$ = $Rs2_i$)
then Rd_i <-$_{32}$ 1
else Rd_i <-$_{32}$ 0

VCMPGTFP (Vector Float Compare Greater Than)

Cada elemento flotante de el registro fuente multimedia Rs1 es comparado con el elemento correspondiente elemento flotante del registro fuente multimedia Rs2, si cada elemento flotante Rs1 es mayor que cada elemento correspondiente de Rs2 el elemento correspondiente de el registro destino multimedia Rd se rellena de unos en caso contrario con ceros.

Formato: VCMPGTFP Rd Rs1 Rs2

Ejemplo: VCMPGTFP m3 m2 m1

Operación:
do i = 0 to 3
If($Rs1_i$ > $Rs2_i$)
then Rd_i <-$_{32}$ 1
else Rd_i <-$_{32}$ 0

VCMPGTSB (Vector Signed Byte Compare Greater Than)

Cada byte con signo del registro fuente multimedia Rs1 es comparado con el byte con signo correspondiente del registro fuente multimedia Rs2, si cada byte con signo Rs1 es mayor que cada elemento correspondiente de Rs2 el byte correspondiente de el registro destino multimedia Rd se rellena de unos en caso contrario con ceros.

Formato: VCMPGTSB Rd Rs1 Rs2

Ejemplo: VCMPGTSB m3 m2 m1

Operación:
do i = 0 to 15
If($Rs1_i$ > $Rs2_i$)
then Rd_i <-$_{8}$ 1
else Rd_i <-$_{8}$ 0

VCMPGTSH (Vector Signed Half Word Compare Greater Than)

Cada media palabra con signo del registro fuente multimedia Rs1 es comparada con la media palabra con signo correspondiente del registro fuente multimedia Rs2, si cada media palabra con signo Rs1 es mayor que cada media palabra correspondiente de Rs2 la media palabra correspondiente de el registro destino multimedia Rd se rellena de unos en caso contrario con ceros.

Formato: VCMPGTSH Rd Rs1 Rs2

Ejemplo: VCMPGTSH m3 m2 m1

Operación: do i = 0 to 7
If($Rs1_i > Rs2_i$)
then $Rd_i <-_{16} 1$
else $Rd_i <-_{16} 0$

VCMPGTSW (Vector Signed Word Compare Greater Than)

Cada palabra con signo del registro fuente multimedia Rs1 es comparada con la palabra con signo correspondiente del registro fuente multimedia Rs2, si cada palabra con signo Rs1 es mayor que cada palabra correspondiente de Rs2 la palabra correspondiente de el registro destino multimedia Rd se rellena de unos en caso contrario con ceros.

Formato: VCMPGTSW Rd Rs1 Rs2

Ejemplo: VCMPGTSW m3 m2 m1

Operación: do i = 0 to 3
If($Rs1_i > Rs2_i$)
then $Rd_i <-_{32} 1$
else $Rd_i <-_{32} 0$

VCMPGTUB (Vector Unsigned Byte Compare Greater Than)

Cada byte sin signo del registro fuente multimedia Rs1 es comparado con el byte sin signo correspondiente del registro fuente multimedia Rs2, si cada byte sin signo Rs1 es mayor que cada elemento correspondiente de Rs2 el byte correspondiente de el registro destino multimedia Rd se rellena de unos en caso contrario con ceros.

Formato: VCMPGTUB Rd Rs1 Rs2

Ejemplo: VCMPGTUB m3 m2 m1

Operación: do i = 0 to 15
If($Rs1_i > Rs2_i$)
then $Rd_i <-_{8} 1$
else $Rd_i <-_{8} 0$

VCMPGTUH (Vector Unsigned Half Word Compare Greater Than)

Cada media palabra sin signo del registro fuente multimedia Rs1 es comparada con la media palabra sin signo correspondiente del registro fuente multimedia Rs2, si cada media palabra sin signo Rs1 es mayor que cada media palabra correspondiente de Rs2 la media palabra correspondiente de el registro destino multimedia Rd se rellena de unos en caso contrario con ceros.

Formato: VCMPGTUH Rd Rs1 Rs2

Ejemplo: VCMPGTUH m3 m2 m1

Operación: do i = 0 to 7
If($Rs1_i > Rs2_i$)
then $Rd_i <-_{16} 1$
else $Rd_i <-_{16} 0$

VCMPGTUW (Vector Unsigned Word Compare Greater Than)

Cada palabra sin signo del registro fuente multimedia Rs1 es comparada con la palabra sin signo correspondiente del registro fuente multimedia Rs2, si cada palabra sin signo Rs1 es mayor que cada palabra correspondiente de Rs2 la palabra correspondiente de el registro destino multimedia Rd se rellena de unos en caso contrario con ceros.

Formato: VCMPGTUW Rd Rs1 Rs2

Ejemplo: VCMPGTUW m3 m2 m1

Operación: do i = 0 to 3
If($Rs1_i > Rs2_i$)
then $Rd_i <-_{32} 1$
else $Rd_i <-_{32} 0$

VCTUX (Vector Convert to Unsigned Fixed-Point Word Saturated)

Cada palabra entera sin signo de el registro destino multimedia Rd es el resultado multiplicar cada elemento flotante del registro destino Rs1 por dos elevado a la potencia del registro destino Rs2, saturando el resultado y sin signo.
Si hay saturación en la multiplicación, el registro VSCR[SAT] se pone a uno.

Formato: VCTUX Rd Rs1 Rs2

Ejemplo: VCTUX m0 m1 #5

Operación: do i = 0 to 3
$Rd_i <-_{32}$ Saturacion ($Rs1_i * 2^{Rs2}$)

VEXPTEFP (Vector Is 2 Raised to the Exponent Estimate Floating-Point)

Cada elemento flotante del registro destino Rd es el resultado de una estimación de dos elevado al correspondiente elemento flotante del registro Rs1.

Formato: VEXPTEFP Rd Rs1

Ejemplo: VCTUX m0 m1

Operación: do i = 0 to 3
$Rd_i <-_{32} 2^{Rs1i}$

VLOGEFP (Vector Log2 Estimate Floating-Point)

Cada elemento flotante de el registro destino multimedia Rd es el resultado de una estimación del logaritmo en base dos de cada elemento flotante del registro fuente multimedia Rs1.

Formato: VLOGEFP Rd Rs1

Ejemplo: VLOGEFP m0 m1

Operación: do i = 0 to 3
$Rd_i <-_{32} Log_2(Rs1_i)$

VMADDFP (Vector Multiply Add)

Cada elemento flotante del registro destino multimedia Rd es la suma de el correspondiente elemento flotante del registro fuente multimedia Rs3 con el producto de los elementos flotantes de los registros fuentes multimedia Rs1 y Rs2.
Si VSCR[NJ] =1 cada operador denormalizado se trunca a cero conservando su signo antes de la operación, si la operación y cada elemento denormalizado del resultado es truncado a cero con el mismo signo.

Formato: VMADDFP Rd Rs1 Rs2 Rs3

Ejemplo: VMADDFP m0 m1 m4 m5

Operación: do i = 0 to 3
$Rd_i <-_{32} (Rs1_i * Rs2_i + Rs3_i)$

VMAXFP (Vector Flotat Maximum)

Cada elemento flotante del registro destino multimedia Rd es el valor más alto de los elementos flotantes correspondientes de los registros fuentes multimedia Rs1 y Rs2.

Formato: VMAXFP Rd Rs1 Rs2

Ejemplo: VMAXFP m0 m1 m4

Operación: do i = 0 to 3
$Rd_i <-_{32} Maximo(Rs1_i,Rs2_i)$

VMAXSB (Vector Signed Byte Maximum)

Cada byte con signo del registro destino multimedia Rd es el valor más alto de los byte con signo correspondientes de los registros fuentes multimedia Rs1 y Rs2.

Formato: VMAXSB Rd Rs1 Rs2

Ejemplo: VMAXSB m0 m1 m4

Operación: do i = 0 to 15
$Rd_i <-_{8} Maximo(Rs1_i,Rs2_i)$

VMAXSH (Vector Signed Half Word Maximum)

Cada media palabra con signo del registro destino multimedia Rd es el valor más alto de las medias palabras con signo correspondientes de los registros fuentes multimedia Rs1 y Rs2.

Formato: VMAXSH Rd Rs1 Rs2

Ejemplo: VMAXSH m0 m1 m4

Operación: do i = 0 to 7
$Rd_i <-_{16} Maximo(Rs1_i,Rs2_i)$

VMAXSW (Vector Signed Word Maximum)

Cada palabra con signo del registro destino multimedia Rd es el valor más alto de las palabras con signo correspondientes de los registros fuentes multimedia Rs1 y Rs2.

Formato: VMAXSW Rd Rs1 Rs2

Ejemplo: VMAXSW m0 m1 m4

Operación: do i = 0 to 3
$Rd_i <-_{32} Maximo(Rs1_i,Rs2_i)$

VMAXUB (Vector Unsigned Byte Maximum)

Cada byte sin signo del registro destino multimedia Rd es el valor más alto de los byte sin signo correspondientes de los registros fuentes multimedia Rs1 y Rs2.

Formato: VMAXUB Rd Rs1 Rs2

Ejemplo: VMAXUB m0 m1 m4

Operación: do i = 0 to 15
$Rd_i <-_8$ Maximo($Rs1_i$,$Rs2_i$)

VMAXUH (Vector Unsigned Half Word Maximum)

Cada media palabra sin signo del registro destino multimedia Rd es el valor más alto de las medias palabras sin signo correspondientes de los registros fuentes multimedia Rs1 y Rs2.

Formato: VMAXUH Rd Rs1 Rs2

Ejemplo: VMAXUH m0 m1 m4

Operación: do i = 0 to 7
$Rd_i <-_{16}$ Maximo($Rs1_i$,$Rs2_i$)

VMAXUW (Vector Unsigned Word Maximum)

Cada palabra sin signo del registro destino multimedia Rd es el valor más alto de las palabras sin signo correspondientes de los registros fuentes multimedia Rs1 y Rs2.

Formato: VMAXUW Rd Rs1 Rs2

Ejemplo: VMAXUW m0 m1 m4

Operación: do i = 0 to 3
$Rd_i <-_{32}$ Maximo($Rs1_i$,$Rs2_i$)

VMHADDSHS (Vector Multiply Add Saturated)

Cada media palabra con signo del registro destino multimedia Rd es la suma de la correspondiente media palabra del registro fuente multimedia Rs3 con los 17 bits más altos de la multiplicación de los correspondientes registros fuentes Rs1 y Rs2.
Si hay saturación, el registro VSCR[SAT] se pone a uno.

Formato: VMHADDSHS Rd Rs1 Rs2 Rs3

Ejemplo: VMHADDSHS m0 m1 m4 m5

Operación: do i = 0 to 7
$Rd_i <-_{16}$ Saturacion $((Rs1_i * Rs2_i)/ 2^{15} + Rs3_i)$

VMHRADDSHS (Vector Multiply Round and Add Saturated)

Cada media palabra con signo de el registro destino multimedia Rd es la suma saturada de la media palabra correspondiente del registro fuente multimedia Rs3 con los 17 bits más altos de la multiplicación de la media palabra de los registros fuente multimedia Rs1 y Rs2. Si hay saturación, el registro VSCR[SAT] se pone a uno.

Formato: VMHRADDSHS Rd Rs1 Rs2 Rs3

Ejemplo: VMHRADDSHS m0 m1 m4 m5

Operación: do i = 0 to 7
$Rd_i <-_{16}$ Saturacion $((Rs1_i * Rs2_i + 2^{14})/ 2^{15} + Rs3_i)$

VMINFP (Vector Flotat Minimun)

Cada elemento flotante del registro destino multimedia Rd es el valor más bajo de los elementos flotantes correspondientes de los registros fuentes multimedia Rs1 y Rs2.

Formato: VMINFP Rd Rs1 Rs2

Ejemplo: VMINFP m0 m1 m4

Operación: do i = 0 to 3
$Rd_i <-_{32}$ Minimo$(Rs1_i, Rs2_i)$

VMINSB (Vector Signed Byte Minimun)

Cada byte con signo del registro destino multimedia Rd es el valor más bajo de los byte con signo correspondientes de los registros fuentes multimedia Rs1 y Rs2.

Formato: VMINSB Rd Rs1 Rs2

Ejemplo: VMINSB m0 m1 m4

Operación: do i = 0 to 15
$Rd_i <-_8$ Minimo($Rs1_i$,$Rs2_i$)

VMINSH (Vector Signed Half Word Minimun)

Cada media palabra con signo del registro destino multimedia Rd es el valor más bajo de las medias palabras con signo correspondientes de los registros fuentes multimedia Rs1 y Rs2.

Formato: VMINSH Rd Rs1 Rs2

Ejemplo: VMINSH m0 m1 m4

Operación: do i = 0 to 7
$Rd_i <-_{16}$ Minimo($Rs1_i$,$Rs2_i$)

VMINSW (Vector Signed Word Minimun)

Cada palabra con signo del registro destino multimedia Rd es el valor más bajo de las palabras con signo correspondientes de los registros fuentes multimedia Rs1 y Rs2.

Formato: VMINSW Rd Rs1 Rs2

Ejemplo: VMINSW m0 m1 m4

Operación: do i = 0 to 3
$Rd_i <-_{32}$ Minimo($Rs1_i$,$Rs2_i$)

VMINUB (Vector Unsigned Byte Minimun)

Cada byte sin signo del registro destino multimedia Rd es el valor más bajo de los byte sin signo correspondientes de los registros fuentes multimedia Rs1 y Rs2.

Formato: VMINUB Rd Rs1 Rs2

Ejemplo: VMINUB m0 m1 m4

Operación: do i = 0 to 15
$Rd_i <-_8$ Minimo($Rs1_i$,$Rs2_i$)

VMINUH (Vector Unsigned Half Word Minimun)

Cada media palabra sin signo del registro destino multimedia Rd es el valor más bajo de las medias palabras sin signo correspondientes de los registros fuentes multimedia Rs1 y Rs2.

Formato: VMINUH Rd Rs1 Rs2

Ejemplo: VMINUH m0 m1 m4

Operación: do i = 0 to 7
$Rd_i <-_{16}$ Minimo($Rs1_i$,$Rs2_i$)

VMINUW (Vector Unsigned Word Minimun)

Cada palabra sin signo del registro destino multimedia Rd es el valor más bajo de las palabras sin signo correspondientes de los registros fuentes multimedia Rs1 y Rs2.

Formato: VMAXUW Rd Rs1 Rs2

Ejemplo: VMAXUW m0 m1 m4

Operación: do i = 0 to 3
$Rd_i <-_{32}$ Minimo($Rs1_i$,$Rs2_i$)

VMLADDUHM (Vector Multiply Low and Add Unsigned Half Word)

Cada media palabra sin signo del registro destino multimedia Rd son los 16 bits más bajos de la suma de cada media palabra sin signo del registro fuente multimedia Rd3 sumado a la multiplicación de cada media palabra sin signo de los registros fuente Rd1 y Rd2.

Formato: VMLADDUHM Rd Rs1 Rs2 Rs3

Ejemplo: VMLADDUHM m0 m1 m4 m5

Operación: do i = 0 to 7
$Rd_i <-_{16}$ ($Rs1_i$ * $Rs2_i$ + $Rs3_i$)

VMRGHB (Vector Byte Merge High)

Los bytes impares del registro destino multimedia Rd son la mitad de los bytes más altos del registro fuente multimedia Rs1. Los bytes pares del registro destino multimedia Rd son la mitad de los elementos más altos del registro fuente multimedia Rs2.

Formato: VMRGHB Rd Rs1 Rs2

Ejemplo: VMRGHB m0 m1 m4

Operación: do i = 0 to 7
$Rd_{2i} <-_{8} Rs1_i$
$Rd_{2i+1} <-_{8} Rs2_i$

VMRGHH (Vector Half Word Merge High)

Las medias palabras impares del registro destino multimedia Rd son la mitad de las medias palabras más altas del registro fuente multimedia Rs1. Las medias palabras pares del registro destino multimedia Rd son la mitad de las medias palabras más altas del registro fuente multimedia Rs2.

Formato: VMRGHH Rd Rs1 Rs2

Ejemplo: VMRGHH m0 m1 m4

Operación: do i = 0 to 3
$Rd_{2i} <-_{16} Rs1_i$
$Rd_{2i+1} <-_{16} Rs2_i$

VMRGHW (Vector Word Merge High)

Las palabras impares del registro destino multimedia Rd son la mitad de las palabras más altas del registro fuente multimedia Rs1. Las palabras pares del registro destino multimedia Rd son la mitad de las palabras más altas del registro fuente multimedia Rs2.

Formato: VMRGHW Rd Rs1 Rs2

Ejemplo: VMRGHW m0 m1 m4

Operación: do i = 0 to 1
$Rd_{2i} <-_{32} Rs1_i$
$Rd_{2i+1} <-_{32} Rs2_i$

VMRGLB (Vector Byte Merge Low)

Los bytes impares del registro destino multimedia Rd son la mitad de los bytes más bajos del registro fuente multimedia Rs1. Los bytes pares del registro destino multimedia Rd son la mitad de los elementos más bajos del registro fuente multimedia Rs2.

Formato: VMRGLB Rd Rs1 Rs2

Ejemplo: VMRGLB m0 m1 m4

Operación: do i = 0 to 7
$Rd_{2i} <-_{8} Rs1_{i+7}$
$Rd_{2i+1} <-_{8} Rs2_{i+7}$

VMRGLH (Vector Half Word Merge Low)

Las medias palabras impares del registro destino multimedia Rd son la mitad de las medias palabras más bajas del registro fuente multimedia Rs1. Las medias palabras pares del registro destino multimedia Rd son la mitad de las medias palabras más bajas del registro fuente multimedia Rs2.

Formato: VMRGLH Rd Rs1 Rs2

Ejemplo: VMRGLH m0 m1 m4

Operación: do i = 0 to 3
$Rd_{2i} <-_{16} Rs1_{i+3}$
$Rd_{2i+1} <-_{16} Rs2_{i+3}$

VMRGLW (Vector Word Merge Low)

Las palabras impares del registro destino multimedia Rd son la mitad de las palabras más bajas del registro fuente multimedia Rs1. Las palabras pares del registro destino multimedia Rd son la mitad de las palabras más bajas del registro fuente multimedia Rs2.

Formato: VMRGLW Rd Rs1 Rs2

Ejemplo: VMRGLW m0 m1 m4

Operación: do i = 0 to 1
$Rd_{2i} <-_{32} Rs1_{i+1}$
$Rd_{2i+1} <-_{32} Rs2_{i+1}$

VMSUMMBM (Vector Signed Byte Multiply Sum)

Cada palabra con signo de el registro destino multimedia Rd es la suma la palabra con signo de el registro fuente multimedia Rs3 con la multiplicación de los bytes con signo de los registros fuente multimedia Rs1 y Rs2 que se superponen a los elementos de Rs3.

Formato: VMSUMMBM Rd Rs1 Rs2

Ejemplo: VMSUMMBM m0 m1 m4

Operación: do i = 0 to 3

$Rd_i <-_{32} (Rs1_{4i} * Rs2_{4i}) + (Rs1_{4I+1} * Rs2_{4i+1}) + (Rs1_{4i+2} * Rs2_{4i+2}) + (Rs1_{4i+3} * Rs2_{4i+3}) + Rs3_i$

VMSUMSHM (Vector Signed Half Word Multiply Sum)

Cada palabra con signo de el registro destino multimedia Rd es la suma dela palabra con signo de el registro fuente multimedia Rs3 con la multiplicación de las medias palabra con signo de los registros fuente multimedia Rs1 y Rs2 que se superponen a los elementos de Rs3.

Formato: VMSUMSHM Rd Rs1 Rs2

Ejemplo: VMSUMSHM m0 m1 m4

Operación: do i = 0 to 3

$Rd_i <-_{32} (Rs1_{2i} * Rs2_{2i}) + (Rs1_{2I+1} * Rs2_{2i+1}) + Rs3_i$

VMSUMSHS (Vector Signed Half Word Multiply Sum Saturated)

Cada palabra con signo de el registro destino multimedia Rd es la suma dela palabra con signo de el registro fuente multimedia Rs3 con la multiplicación de las medias palabra con signo de los registros fuente multimedia Rs1 y Rs2 que se superponen a los elementos de Rs3. Si hay saturación en la suma, el registro VSCR[SAT] se pone a uno.

Formato: VMSUMSHS Rd Rs1 Rs2

Ejemplo: VMSUMSHS m0 m1 m4

Operación: do i = 0 to 3

$Rd_i <-_{32}$ Saturacion $((Rs1_{2i} * Rs2_{2i}) + (Rs1_{2I+1} * Rs2_{2i+1}) + Rs3_i)$

VMSUMUBM (Vector Unsigned Byte Multiply Sum)

Cada palabra sin signo de el registro destino multimedia Rd es la suma la palabra sin signo de el registro fuente multimedia Rs3 con la multiplicación de los bytes sin signo de los registros fuente multimedia Rs1 y Rs2 que se superponen a los elementos de Rs3.

Formato: VMSUMUBM Rd Rs1 Rs2

Ejemplo: VMSUMUBM m0 m1 m4

Operación: do i = 0 to 3

$Rd_i <-_{32} (Rs1_{4i} * Rs2_{4i}) + (Rs1_{4i+1} * Rs2_{4i+1}) + (Rs1_{4i+2} * Rs2_{4i+2}) + (Rs1_{4i+3} * Rs2_{4i+3}) + Rs3_i$

VMSUMUHM (Vector Unsigned Half Word Multiply Sum)

Cada palabra sin signo de el registro destino multimedia Rd es la suma la palabra sin signo de el registro fuente multimedia Rs3 con la multiplicación de las medias palabra sin signo de los registros fuente multimedia Rs1 y Rs2 que se superponen a los elementos de Rs3.

Formato: VMSUMUHM Rd Rs1 Rs2

Ejemplo: VMSUMUHM m0 m1 m4

Operación: do i = 0 to 3

$Rd_i <-_{32} (Rs1_{2i} * Rs2_{2i}) + (Rs1_{2i+1} * Rs2_{2i+1}) + Rs3_i$

VMSUMUHS (Vector Unsigned Half Word Multiply Sum Saturated)

Cada palabra sin signo de el registro destino multimedia Rd es la suma la palabra sin signo de el registro fuente multimedia Rs3 con la multiplicación de las medias palabra sin signo de los registros fuente multimedia Rs1 y Rs2 que se superponen a los elementos de Rs3. Si hay saturación en la suma, el registro VSCR[SAT] se pone a uno.

Formato: VMSUMUHS Rd Rs1 Rs2

Ejemplo: VMSUMUHS m0 m1 m4

Operación: do i = 0 to 3

$Rd_i <-_{32}$ Saturacion $((Rs1_{2i} * Rs2_{2i}) + (Rs1_{2i+1} * Rs2_{2i+1}) + Rs3_i)$

VMULESB (Vector Signed Byte Multiply Even)

Cada media palabra con signo del registro destino multimedia Rd es la multiplicación de los bytes con signo pares de los registros fuente multimedia Rs1 y Rs2.

Formato: VMULESB Rd Rs1 Rs2

Ejemplo: VMULESB m0 m1 m4

Operación: do i = 0 to 7
$Rd_i <-_{16} Rs1_{2i} * Rs2_{2i}$

VMULESH (Vector Signed Half Word Multiply Even)

Cada palabra con signo del registro destino multimedia Rd es la multiplicación de las medias palabras con signo pares de los registros fuente multimedia Rs1 y Rs2.

Formato: VMULESH Rd Rs1 Rs2

Ejemplo: VMULESH m0 m1 m4

Operación: do i = 0 to 3
$Rd_i <-_{32} Rs1_{2i} * Rs2_{2i}$

VMULEUB (Vector Unsigned Byte Multiply Even)

Cada media palabra sin signo del registro destino multimedia Rd es la multiplicación de los bytes sin signo pares de los registros fuente multimedia Rs1 y Rs2.

Formato: VMULEUB Rd Rs1 Rs2

Ejemplo: VMULEUB m0 m1 m4

Operación: do i = 0 to 7
$Rd_i <-_{16} Rs1_{2i} * Rs2_{2i}$

VMULEUH (Vector Unsigned Half Word Multiply Even)

Cada palabra sin signo del registro destino multimedia Rd es la multiplicación de las medias palabras sin signo pares de los registros fuente multimedia Rs1 y Rs2.

Formato: VMULEUH Rd Rs1 Rs2

Ejemplo: VMULEUH m0 m1 m4

Operación: do i = 0 to 3
$Rd_i <-_{32} Rs1_{2i} * Rs2_{2i}$

VMULOSB (Vector Signed Byte Multiply Odd)

Cada media palabra con signo del registro destino multimedia Rd es la multiplicación de los bytes con signo impares de los registros fuente multimedia Rs1 y Rs2.

Formato: VMULOSB Rd Rs1 Rs2

Ejemplo: VMULOSB m0 m1 m4

Operación: do i = 0 to 7
$Rd_i <-_{16} Rs1_{2i+1} * Rs2_{2i+1}$

VMULOSH (Vector Signed Half Word Multiply Odd)

Cada palabra con signo del registro destino multimedia Rd es la multiplicación de las medias palabras con signo impares de los registros fuente multimedia Rs1 y Rs2.

Formato: VMULOSH Rd Rs1 Rs2

Ejemplo: VMULOSH m0 m1 m4

Operación: do i = 0 to 3
$Rd_i <-_{32} Rs1_{2i+1} * Rs2_{2i+1}$

VMULOUB (Vector Unsigned Byte Multiply Odd)

Cada media palabra sin signo del registro destino multimedia Rd es la multiplicación de los bytes sin signo impares de los registros fuente multimedia Rs1 y Rs2.

Formato: VMULOUB Rd Rs1 Rs2

Ejemplo: VMULOUB m0 m1 m4

Operación: do i = 0 to 7
$Rd_i <-_{16} Rs1_{2i+1} * Rs2_{2i+1}$

VMULOUH (Vector Unsigned Half Word Multiply Odd)

Cada palabra sin signo del registro destino multimedia Rd es la multiplicación de las medias palabras sin signo impares de los registros fuente multimedia Rs1 y Rs2.

Formato: VMULOUH Rd Rs1 Rs2

Ejemplo: VMULOUH m0 m1 m4

Operación: do i = 0 to 3
$Rd_i <-_{32} Rs1_{2i+1} * Rs2_{2i+1}$

VNMSUBFP (Vector Negative Multiply Subtract)

Cada elemento flotante de el registro destino multimedia Rd es la resta de la multiplicación de los elementos flotantes de los registros fuente multimedia Rs1 y Rs2 menos el elemento flotante de el registro fuente multimedia Rs3.
Si VSCR[NJ] =1 cada operador denormalizado se trunca a cero conservando su signo antes de la operación, si la operación y cada elemento denormalizado del resultado es truncado a cero con el mismo signo.

Formato: VNMSUBFP Rd Rs1 Rs2 Rs3

Ejemplo: VMULOUH m0 m1 m4 m7

Operación: do i = 0 to 3
$Rd_i <-_{32} Rs1_i * Rs2_i - Rs3$

VNOR (Vector Logical NOR)

Cada bit del registro destino multimedia Rd es la operación lógica NOR de cada uno de los bits de el registro fuente multimedia Rs1 con cada uno de los bits de el registro fuente multimedia Rs2.

Formato: VNOR Rd Rs1 Rs2

Ejemplo: VNOR m3 m2 m1

Operacion: $Rd <-_{128}$ Rs1 | Rs2)

VOR (Vector Logical OR)

Cada bit de el registro destino multimedia Rd es la operación lógica OR de cada uno de los bits de el registro fuente multimedia Rs1 con cada uno de los bits de el registro fuente multimedia Rs2.

Formato: VOR Rd Rs1 Rs2

Ejemplo: VOR m3 m2 m1

Operacion: $Rd <-_{128}$ Rs1 | Rs2

VPERM (Vector Permute)

Cada elemento de el registro destino multimedia Rd es seleccionado por los elementos de el registro fuente Rs3 de los elementos de los registros fuentes Rs1 y Rs2

Formato: VPERM Rd Rs1 Rs2 Rs3

Ejemplo: VPERM m3 m2 m1 m4

Operacion: do i = 0 to 15
$j <-_4 Rs3_i[4\text{-}7]$
if $Rs3_i[3] = 0$
then $Rd_i <-_8 Rs1_j$
else $Rd_i <-_8 Rs2_j$

VPKPX (Vector Pack Pixel)

Cada elemento de la parte alta de el registro fuente multimedia Rd es el empaquetamiento de cada pixel de el registro fuente multimedia Rs1. Cada elemento de la parte baja de el registro fuente multimedia Rd es el empaquetamiento de cada pixel de el registro fuente multimedia Rs2.

Formato: VPKPX Rd Rs1 Rs2

Ejemplo: VPKPX m3 m2 m1

Operacion: do i = 0 to 3
$Rd_i <-_{32} Rs1_i [7] || Rs1_i [8\text{-}12] || Rs1_i [16\text{-}20] || Rs1_i [24\text{-}28]$
$Rd_{i+4} <-_{32} Rs2_i [7] || Rs2_i [8\text{-}12] || Rs2_i [16\text{-}20] || Rs2_i [24\text{-}28]$

VPKSHSS (Vector Signed Half Word Pack Saturated)

Cada byte con signo de la parte alta de el registro fuente multimedia Rd es la saturación de cada media palabra con signo de el registro fuente Rs1. Cada byte con signo de la parte baja de el registro fuente Rd es la saturación de cada media palabra con signo del registro fuente Rs2. Si hay saturación en la suma, el registro VSCR[SAT] se pone a uno.

Formato: VPKSHSS Rd Rs1 Rs2

Ejemplo: VPKSHSS m3 m2 m1

Operacion: do i = 0 to 7
$Rd_i <-_8$ Saturacion($Rs1_i$)
$Rd_{i+8} <-_8$ Saturacion($Rs2_i$)

VPKSHUS (Vector Unsigned Half Word Pack Saturated Unsigned)

Cada byte sin signo de la parte alta de el registro fuente multimedia Rd es el truncamiento de cada media palabra sin signo del registro fuente multimedia Rs1. Cada byte sin signo de la parte baja de el registro fuente multimedia Rd es el truncamiento de cada media palabra sin signo del registro fuente multimedia Rs2.

Formato: VPKUHUS Rd Rs1 Rs2

Ejemplo: VPKUHUS m3 m2 m1

Operacion: do i = 0 to 7
$Rd_i <-_8$ Truncamiento($Rs1_{i,}$,8)
$Rd_{i+8} <-_8$ Truncamiento($Rs2_i$, 8)

VPKSWSS (Vector Signed Word Pack Saturated)

Cada media palabra con signo de la parte alta de el registro fuente multimedia Rd es la saturación de cada palabra con signo de el registro fuente Rs1. Cada media palabra con signo de la parte baja de el registro fuente Rd es la saturación de cada palabra con signo del registro fuente Rs2. Si hay saturación en la suma, el registro VSCR[SAT] se pone a uno.

Formato: VPKSWSS Rd Rs1 Rs2

Ejemplo: VPKSWSS m3 m2 m1

Operacion: do i = 0 to 3
$Rd_i <-_{16}$ Saturacion($Rs1_i$)
$Rd_{i+4} <-_{16}$ Saturacion($Rs2_i$)

VPKSWUS (Vector Signed Word Pack Saturated Unsigned)

Cada media palabra sin signo de la parte alta de el registro fuente multimedia Rd es la saturación de cada palabra con signo de el registro fuente Rs1. Cada media palabra sin signo de la parte baja de el registro fuente Rd es la saturación de cada palabra con signo del registro fuente Rs2. Si hay saturación en la suma, el registro VSCR[SAT] se pone a uno.

Formato: VPKSWUS Rd Rs1 Rs2

Ejemplo: VPKSWUS m3 m2 m1

Operacion: do i = 0 to 3
$Rd_i <-_{16}$ Saturacion ($Rs1_i$)
$Rd_{i+4} <-_{16}$ Saturacion ($Rs2_i$)

VPKUHUM (Vector Unsigned Half Word Pack)

Cada media palabra sin signo de la parte alta de el registro fuente multimedia Rd es el truncamiento de cada palabra sin signo del registro fuente multimedia Rs1. Cada media palabra sin signo de la parte baja de el registro fuente multimedia Rd es el truncamiento de cada palabra sin signo del registro fuente multimedia Rs2.

Formato: VPKUHUM Rd Rs1 Rs2

Ejemplo: VPKUHUM m3 m2 m1

Operacion: do i = 0 to 3
$Rd_i <-_{16}$ Truncamiento($Rs1_{i,}$,16)
$Rd_{i+8} <-_{16}$ Truncamiento($Rs2_i$, 16)

VPKUHUS (Vector Unsigned Half Word Pack Saturated)

Cada byte sin signo de la parte alta de el registro fuente multimedia Rd es la saturación de cada media palabra sin signo de el registro fuente Rs1. Cada byte sin signo de la parte baja de el registro fuente Rd es la saturación de cada media palabra sin signo del registro fuente Rs2. Si hay saturación en la suma, el registro VSCR[SAT] se pone a uno.

Formato: VPKUHUS Rd Rs1 Rs2

Ejemplo: VPKUHUS m3 m2 m1

Operacion: do i = 0 to 7
$Rd_i <-_{8}$ Saturacion ($Rs1_i$)
$Rd_{i+8} <-_{8}$ Saturacion ($Rs2_i$)

VPKUWUM (Vector Unsigned Word Pack)

Cada media palabra sin signo de la parte alta de el registro fuente multimedia Rd es la saturación de cada palabra sin signo de el registro fuente Rs1. Cada media palabra sin signo de la parte baja de el registro fuente Rd es la saturación de cada palabra sin signo del registro fuente Rs2. Si hay saturación en la suma, el registro VSCR[SAT] se pone a uno.

Formato: VPKUWUM Rd Rs1 Rs2

Ejemplo: VPKUWUM m3 m2 m1

Operacion: do i = 0 to 3
$Rd_i <-_{16}$ Saturacion ($Rs1_i$)
$Rd_{i+4} <-_{16}$ Saturacion ($Rs2_i$)

VPKUWUS (Vector Unsigned Word Pack Saturated Unsigned)

Cada media palabra sin signo de la parte alta de el registro fuente multimedia Rd es la saturación de cada palabra sin signo de el registro fuente Rs1. Cada media palabra sin signo de la parte baja de el registro fuente Rd es la saturación de cada palabra sin signo del registro fuente Rs2.
Si hay saturación en la suma, el registro VSCR[SAT] se pone a uno.

Formato: VPKUWUS Rd Rs1 Rs2

Ejemplo: VPKUWUS m3 m2 m1

Operacion: do i = 0 to 3
$Rd_i <-_{16}$ Saturacion ($Rs1_i$)
$Rd_{i+4} <-_{16}$ Saturacion ($Rs2_i$)

VREFP (Vector Reciprocal Estimate)

Cada elemento flotante del registro destino multimedia Rd es el reciproco de cada elemento flotante de el registro fuente multimedia Rs1.
Si VSCR[NJ] =1 cada operador denormalizado se trunca a cero conservando su signo antes de la operación, si la operación y cada elemento denormalizado del resultado es truncado a cero con el mismo signo.

Formato: VREFP Rd Rs1

Ejemplo: VREFP m3 m2

Operacion: do i = 0 to 3
$Rd_i <-_{32}$ Reciproco($Rs1_i$)

VRFIM (Vector Floor)

Cada elemento flotante del registro destino multimedia Rd es el redondeo al valor suelo de cada elemento flotante del registro fuente multimedia Rs1.
Si VSCR[NJ] =1 cada operador denormalizado se trunca a cero conservando su signo antes de la operación.

Formato: VRFIM Rd Rs1

Ejemplo: VRFIM m3 m2

Operacion: do i = 0 to 3
$Rd_i <-_{32}$ Suelo($Rs1_i$)

VRFIN (Vector Round)

Cada elemento flotante del registro destino multimedia Rd es el redondeo de cada elemento flotante del registro fuente multimedia Rs1 al entero más cercano. Si ambos valores se encuentran a la misma distancia el redondeo se hace al par. La operación es independiente de VSCR[NJ].

Formato: VRFIN Rd Rs1

Ejemplo: VRFIN m3 m2

Operacion: do i = 0 to 3
$Rd_i <-_{32}$ Cercano($Rs1_i$)

VRFIP (Vector Ceiling)

Cada elemento flotante del registro destino multimedia Rd es el redondeo de cada elemento flotante del registro fuente multimedia Rs1 al entero mayor o igual que el elemento.

Formato: VRFIP Rd Rs1

Ejemplo: VRFIP m3 m2

Operacion: do i = 0 to 3
$Rd_i <-_{32}$ Ceil($Rs1_i$)

VRFIZ (Vector Truncate)

Cada elemento flotante del registro destino multimedia Rd es el redondeo de cada elemento flotante del registro fuente multimedia Rs1 quedándose con la parte entera del elemento.
La operación es independiente de VSCR[NJ].

Formato: VRFIZ Rd Rs1

Ejemplo: VRFIZ m3 m2

Operacion: do i = 0 to 3
$Rd_i <-_{32}$ ($Rs1_i$)

VRLB (Vector Byte Rotate Left)

Cada byte de el registro destino multimedia Rd es la rotación de cada byte del registro fuente multimedia Rs1 tantas veces como indique cada elemento del registro fuente Rs2.

Formato: VRLB Rd Rs1 Rs2

Ejemplo: VRLB m3 m2 m1

Operacion: do i = 0 to 15
$Rd_i <-_8$ Rotar($Rs1_i$,$Rs2_i$)

VRLH (Vector Half Word Rotate Left)

Cada media palabra de el registro destino multimedia Rd es la rotación de cada media palabra del registro fuente multimedia Rs1 tantas veces como indique cada elemento del registro fuente Rs2.

Formato: VRLH Rd Rs1 Rs2

Ejemplo: VRLH m3 m2 m1

Operacion: do i = 0 to 7
$Rd_i <-_{16}$ Rotar($Rs1_i$,$Rs2_i$)

VRLW (Vector Word Rotate Left)

Cada palabra de el registro destino multimedia Rd es la rotación de cada palabra del registro fuente multimedia Rs1 tantas veces como indique cada elemento del registro fuente Rs2.

Formato: VRLW Rd Rs1 Rs2

Ejemplo: VRLW m3 m2 m1

Operacion: do i = 0 to 3
$Rd_i <-_{32}$ Rotar($Rs1_i$,$Rs2_i$)

VRSQRTEFP (Vector Reciprocal Square Root Estimate)

Cada elemento flotante del registro destino multimedia Rd es una estimación del reciproco de la raíz cuadrada de cada elemento flotante del registro fuente multimedia Rs1. Si VSCR[NJ] =1 cada operador denormalizado se trunca a cero conservando su signo antes de la operación, si la operación y cada elemento denormalizado del resultado es truncado a cero con el mismo signo.

Formato: VRSQRTEFP Rd Rs1

Ejemplo: VRSQRTEFP m3 m2

Operacion: do i = 0 to 3
$Rd_i <-_{32}$ ReciprocoRaiz(Rs1)

VSEL (Vector Select)

Cada bit de el registro destino multimedia Rd es el correspondiente bit de el registro fuente Rs1 si el bit correspondiente del registro fuente multimedia Rs3 es 0, en caso contrario el bit correspondiente es el del registro fuente Rs2.

Formato: VSEL Rd Rs1 Rs2

Ejemplo: VSEL m3 m2 m0

Operacion: do i = 0 to 127
If $Rs3_i = 0$
then $Rd_i <-_1 Rs1_i$
else $Rd_i <-_1 Rs2_i$

VSL (Vector Shift Left Long)

El resultado de el registro destino multimedia Rd es el desplazamiento a la izquierda del registro fuente multimedia Rs1 el numero de bits especificados en los últimos 3 bits de el último elemento de el registro fuente Rs2.

Formato: VSL Rd Rs1 Rs2

Ejemplo: VSL m3 m2 m0

Operacion: $m <-_1$ Rs2[125-127]
Rd <- DesplazamientoIzquierda(Rs1, m)

VSLB (Vector Byte Shift Left)

Cada byte del registro destino multimedia Rd es el correspondiente byte del registro fuente Rs1 desplazado a la izquierda el número de bits correspondientes al elemento del registro fuente Rs2.

Formato: VSLB Rd Rs1 Rs2

Ejemplo: VSLB m3 m2 m0

Operacion: do i = 0 to 15
$Rd_i <-_8$ DesplazamientoIzquierda ($Rs1_i$, $Rs2_i$)

VSLH (Vector Half Word Shift Left)

Cada media palabra del registro destino multimedia Rd es la correspondiente media palabra del registro fuente Rs1 desplazado a la izquierda el número de bits correspondientes al elemento del registro fuente Rs2.

Formato: VSLH Rd Rs1 Rs2

Ejemplo: VSLH m3 m2 m0

Operacion: do i = 0 to 7
$Rd_i <-_{16}$ DesplazamientoIzquierda ($Rs1_i$, $Rs2_i$)

VSLO (Vector Shift Left by Octet)

El resultado del registro destino multimedia Rd es el desplazamiento a la izquierda de los bytes del registro fuente multimedia Rs1 el número de bytes especificados en los bits [1-4] del ultimo byte del registro fuente Rs2.

Formato: VSLO Rd Rs1 Rs2

Ejemplo: VSLO m3 m2 m0

Operacion: m <- $Rs2_{15}$ [1-4]
do i = 0 to 15
j <- i + m
if j < 16
then $Rd_i <-_8 Rs1_j$
else $Rd_i <-_8 0$

VSLW (Vector Half Word Shift Left)

Cada palabra del registro destino multimedia Rd es la correspondiente palabra del registro fuente Rs1 desplazado a la izquierda el número de bits correspondientes al elemento del registro fuente Rs2.

Formato: VSLW Rd Rs1 Rs2

Ejemplo: VSLW m3 m2 m0

Operacion: do i = 0 to 3
$Rd_i <-_{32}$ DesplazamientoIzquierda ($Rs1_i$, $Rs2_i$)

VSR (Vector Shift Right Long)

El resultado del registro destino multimedia Rd es el desplazamiento a la derecha del registro fuente multimedia Rs1 el número de bits especificados en los últimos 3 bits del último elemento del registro fuente multimedia Rs2.

Formato: VSR Rd Rs1 Rs2

Ejemplo: VSR m3 m2 m0

Operacion: $m <-_1$ Rs2[125-127]
Rd <- DesplazamientoDerecha (Rs1, m)

VSRAB (Vector Byte Shift Right Algebraic)

Cada byte del registro destino multimedia Rd es el correspondiente byte del registro fuente multimedia Rs1 desplazado a la derecha el número de bits correspondientes al elemento del registro fuente multimedia Rs2 extendiendo el signo.

Formato: VSRAB Rd Rs1 Rs2

Ejemplo: VSRAB m3 m2 m0

Operacion: do i = 0 to 15
$Rd_i <-_8$ ExtensionDeSigno(DesplazamientoDerecha ($Rs1_i$, Modulo($Rs2_i$,8)))

VSRAH (Vector Half Word Shift Right Algebraic)

Cada media palabra del registro destino multimedia Rd es la correspondiente media palabra del registro fuente multimedia Rs1 desplazado a la derecha el número de bits correspondientes al elemento del registro fuente multimedia Rs2 extendiendo el signo.

Formato: VSRAH Rd Rs1 Rs2

Ejemplo: VSRAH m3 m2 m0

Operacion: do i = 0 to 7
$Rd_i <-_{16}$ ExtensionDeSigno(DesplazamientoDerecha($Rs1_i$, Modulo($Rs2_i$,16)))

VSRAW (Vector Word Shift Right Algebraic)

Cada palabra del registro destino multimedia Rd es la correspondiente palabra del registro fuente multimedia Rs1 desplazado a la derecha el número de bits correspondientes al elemento del registro fuente multimedia Rs2 extendiendo el signo.

Formato: VSRAW Rd Rs1 Rs2

Ejemplo: VSRAW m3 m2 m0

Operacion: do i = 0 to 3
$Rd_i <-_{32}$ ExtensionDeSigno(DesplazamientoDerecha($Rs1_i$, Modulo($Rs2_i$,32)))

VSRB (Vector Byte Shift Right)

Cada byte del registro destino multimedia Rd es el correspondiente byte del registro fuente multimedia Rs1 desplazado a la derecha el número de bits correspondientes al elemento del registro fuente multimedia Rs2.

Formato: VSRB Rd Rs1 Rs2

Ejemplo: VSRB m3 m2 m0

Operacion: do i = 0 to 15
$Rd_i <-_{8}$ DesplazamientoDerecha($Rs1_i$, Modulo($Rs2_i$,8))

VSRH (Vector Half Word Shift Right)

Cada media palabra del registro destino multimedia Rd es la correspondiente media palabra del registro fuente multimedia Rs1 desplazado a la derecha el número de bits correspondientes al elemento del registro fuente multimedia Rs2.

Formato: VSRH Rd Rs1 Rs2

Ejemplo: VSRH m3 m2 m0

Operacion: do i = 0 to 7
$Rd_i <-_{16}$ DesplazamientoDerecha ($Rs1_i$, Modulo($Rs2_i$,16))

VSRO (Vector Shift Right by Octet)

El resultado del registro destino multimedia Rd es el desplazamiento a la derecha de los bytes del registro fuente multimedia Rs1 el número de bytes especificados en los bits [1-4] del ultimo byte del registro fuente Rs2.

Formato: VSRO Rd Rs1 Rs2

Ejemplo: VSRO m3 m2 m0

Operacion: m <- $Rs2_{15}$ [1-4]
do i = 0 to 15
j <- i - m
if j ≥ 0
then $Rd_i <-_8 Rs1_j$
else $Rd_i <-_8 0$

VSRW (Vector Half Word Shift Right)

Cada palabra del registro destino multimedia Rd es la correspondiente palabra del registro fuente multimedia Rs1 desplazado a la derecha el número de bits correspondientes al elemento del registro fuente multimedia Rs2.

Formato: VSRW Rd Rs1 Rs2

Ejemplo: VSRW m3 m2 m0

Operacion: do i = 0 to 3
$Rd_i <-_{16}$ DesplazamientoDerecha ($Rs1_i$, Modulo($Rs2_i$,32)

VSUBCUW (Vector Subtract Carryout)

Cada palabra del registro destino multimedia Rd es un uno extendido de ceros si cada palabra del registro fuente multimedia Rs1 es mayor que la palabra correspondiente del registro fuente Rs2 en caso contrario le corresponden treinta y dos ceros.

Formato: VSUBCUW Rd Rs1 Rs2

Ejemplo: VSUBCUW m3 m2 m0

Operacion: do i = 0 to 3
$Rd[0\text{-}30]_i <-_{31} 0$
If ($Rs1_i > Rs2_i$)
then $Rd[31]_i <-1$
else $Rd[31]_i <- 0$

VSUBFP (Vector Float Subtract)

Se resta cada elemento flotante del registro fuente multimedia Rs1 con el correspondiente elemento flotante del registro fuente multimedia Rs2, guardando el resultado en el elemento correspondiente del registro destino multimedia Rd.
Si VSCR[NJ] =1 cada operador denormalizado se trunca a cero conservando su signo antes de la operación, si la operación y cada elemento denormalizado del resultado es truncado a cero con el mismo signo.

Formato: VSUBFP Rd Rs1 Rs2

Ejemplo: VSUBFP m3 m2 m1

Operacion: do i=0 to 3
$Rd_i <-_{32} Rs1_i - Rs2_i$

VSUBSBS (Vector Signed Byte Subtract Saturated)

Se resta cada byte con signo del registro fuente multimedia Rs1 con el correspondiente byte con signo del registro fuente multimedia Rs2, guardando el resultado en el elemento correspondiente del registro destino multimedia Rd.
Si hay saturación en la resta, el registro VSCR[SAT] se pone a uno.

Formato: VSUBSBS Rd Rs1 Rs2

Ejemplo: VSUBSBS m3 m2 m1

Operacion: do i=0 to 15
$Rd_i <-_8$ Saturacion($Rs1_i - Rs2_i$)

VSUBSHS (Vector Signed Half Word Subtract Saturated)

Se resta cada media palabra con signo del registro fuente multimedia Rs1 con la correspondiente media palabra con signo del registro fuente multimedia Rs2, guardando el resultado en el elemento correspondiente del registro destino multimedia Rd.
Si hay saturación en la resta, el registro VSCR[SAT] se pone a uno.

Formato: VSUBSHS Rd Rs1 Rs2

Ejemplo: VSUBSHS m3 m2 m1

Operación: do i=0 to 7
$Rd_i <-_{16}$ Saturacion ($Rs1_i$ - $Rs2_i$)

VSUBSWS (Vector Signed Word Subtract Saturated)

Se resta cada palabra con signo del registro fuente multimedia Rs1 con la correspondiente palabra con signo del registro fuente multimedia Rs2, guardando el resultado en el elemento correspondiente del registro destino multimedia Rd.
Si hay saturación en la resta, el registro VSCR[SAT] se pone a uno.

Formato: VSUBSWS Rd Rs1 Rs2

Ejemplo: VSUBSWS m3 m2 m1

Operacion: do i=0 to 3
$Rd_i <-_{32}$ Saturacion ($Rs1_i$ - $Rs2_i$)

VSUBUBM (Vector Byte Subtract)

Se resta cada byte del registro fuente multimedia Rs1 con el correspondiente byte del registro fuente multimedia Rs2, guardando el resultado en el elemento correspondiente del registro destino multimedia Rd.

Formato: VSUBUBM Rd Rs1 Rs2

Ejemplo: VSUBUBM m3 m2 m1

Operacion: do i=0 to 15
$Rd_i <-_{8}$ $Rs1_i$ - $Rs2_i$

VSUBUBS (Vector Unsigned Byte Subtract Saturated)

Se resta cada byte sin signo del registro fuente multimedia Rs1 con el correspondiente byte sin signo del registro fuente multimedia Rs2, guardando el resultado en el elemento correspondiente del registro destino multimedia Rd.
Si hay saturación en la resta, el registro VSCR[SAT] se pone a uno.

Formato: VSUBUBS Rd Rs1 Rs2

Ejemplo: VSUBUBS m3 m2 m1

Operacion: do i=0 to 15
$Rd_i <-_8$ Saturacion ($Rs1_i$ - $Rs2_i$)

VSUBUHM (Vector Half Word Subtract)

Se resta cada media palabra del registro fuente multimedia Rs1 con la correspondiente media palabra del registro fuente multimedia Rs2, guardando el resultado en el elemento correspondiente del registro destino multimedia Rd.

Formato: VSUBUHM Rd Rs1 Rs2

Ejemplo: VSUBUHM m3 m2 m1

Operacion: do i=0 to 7
$Rd_i <-_{16}$ $Rs1_i$ - $Rs2_i$

VSUBUHS (Vector Signed Half Word Subtract Saturated)

Se resta cada media palabra sin signo del registro fuente multimedia Rs1 con la correspondiente media palabra sin signo del registro fuente multimedia Rs2, guardando el resultado en el elemento correspondiente del registro destino multimedia Rd.
Si hay saturación en la resta, el registro VSCR[SAT] se pone a uno.

Formato: VSUBUHS Rd Rs1 Rs2

Ejemplo: VSUBUHS m3 m2 m1

Operacion: do i=0 to 7
$Rd_i <-_{16}$ Saturacion ($Rs1_i$ - $Rs2_i$)

VSUBUWM (Vector Word Subtract)

Se resta cada palabra del registro fuente multimedia Rs1 con la correspondiente palabra del registro fuente multimedia Rs2, guardando el resultado en el elemento correspondiente del registro destino multimedia Rd.

Formato: VSUBUWM Rd Rs1 Rs2

Ejemplo: VSUBUWM m3 m2 m1

Operacion: do i=0 to 3
$Rd_i <-_{32} Rs1_i - Rs2_i$

VSUBUWS (Vector Signed Word Subtract Saturated)

Se resta cada palabra sin signo del registro fuente multimedia Rs1 con la correspondiente palabra con sin del registro fuente multimedia Rs2, guardando el resultado en el elemento correspondiente del registro destino multimedia Rd.
Si hay saturación en la resta, el registro VSCR[SAT] se pone a uno.

Formato: VSUBUWS Rd Rs1 Rs2

Ejemplo: VSUBUWS m3 m2 m1

Operacion: do i=0 to 3
$Rd_i <-_{32}$ Saturacion ($Rs1_i - Rs2_i$)

VSUMSWS (Vector Sum Saturated)

Los tres primeros enteros con signo del registro destino multimedia Rd son cero. El cuarto entero con signo del registro destino es la suma de todos los enteros con signo del registro fuente multimedia Rs1 y el cuarto entero con signo del registro fuente multimedia Rs2.
Si hay saturación en la resta, el registro VSCR[SAT] se pone a uno.

Formato: VSUMSWS Rd Rs1 Rs2

Ejemplo: VSUBUWS m3 m2 m1

Operacion: do i=0 to 2
$Rd_i <-_{32} 0$
$Rd_i <-_{32}$ Saturacion ($Rs1_0 + Rs1_1 + Rs1_2 + Rs1_3 + Rs2_3$)

VSUM2SWS (Vector Sum Across Partial (1/2) Saturated)

El primer y el tercer entero consigno del registro destino multimedia Rd son cero. El segundo entero con signo es el resultado de la suma saturada de los dos primeros enteros con signo del primer registro fuente multimedia Rs1 y el segundo entero con signo del registro fuente Rs2. El cuarto entero con signo es el resultado de la suma saturada de los últimos enteros con signo del primer registro fuente multimedia Rs1 y el cuarto entero con signo del registro fuente multimedia Rs2.
Si hay saturación en la resta, el registro VSCR[SAT] se pone a uno.

Formato: VSUM2SWS Rd Rs1 Rs2

Ejemplo: VSUM2SWS m3 m2 m1

Operacion: do i=0 to 1
$Rd_{2i} <-_{32} 0$
$Rd_{2i+1} <-_{32}$ Saturacion ($Rs1_{2i} + Rs1_{2i+1} + Rs2_{2i+1}$)

VSUM4SBS (Vector Byte Signed Sum Across Partial (1/4) Saturated)

Cada palabra con signo del registro destino multimedia Rd es la suma saturada de cada palabra con signo correspondiente al registro fuejnte Rs2 sumados a los bytes con signo del registro fuente Rs1 que se superponen.
Si hay saturación en la resta, el registro VSCR[SAT] se pone a uno.

Formato: VSUM4SBS Rd Rs1 Rs2

Ejemplo: VSUM4SBS m3 m2 m1

Operacion: do i=0 to 3
$Rd_i <-_{32}$ Saturacion ($Rs1_{4i} + Rs1_{4i+1} + Rs1_{4i+2} + Rs1_{4i+3} + Rs2_i$)

VSUM4SHS (Vector Half Word Signed Sum Across Partial (1/4) Saturated)

Cada palabra con signo del registro destino multimedia Rd es la suma saturada de cada palabra con signo correspondiente al registro fuente Rs2 sumados a las medias palabras con signo del registro fuente Rs1 que se superponen.
Si hay saturación en la resta, el registro VSCR[SAT] se pone a uno.

Formato: VSUM4SHS Rd Rs1 Rs2

Ejemplo: VSUM4SHS m3 m2 m1

Operacion: do i=0 to 3
$Rd_i <-_{32}$ Saturacion ($Rs1_{2i} + Rs1_{2i+1} + Rs2_i$)

VSUM4UBS (Vector Byte Unsigned Sum Across Partial (1/4) Saturated)

Cada palabra sin signo del registro destino multimedia Rd es la suma saturada de cada palabra sin signo correspondiente al registro fuente Rs2 sumados a los bytes sin signo del registro fuente Rs1 que se superponen.
Si hay saturación en la resta, el registro VSCR[SAT] se pone a uno.

Formato: VSUM4UBS Rd Rs1 Rs2

Ejemplo: VSUM4UBS m3 m2 m1

Operacion: do i=0 to 3
$Rd_i <-_{32}$ Saturacion ($Rs1_{4i} + Rs1_{4i+1} + Rs1_{4i+2} + Rs1_{4i+3} + Rs2_i$)

VUPKHPX (Vector Pixel Unpack High Element)

El registro destino multimedia Rd es el resultado de desempaquetar la zona alta del registro fuente multimedia Rs1 extendiendo el signo.

Formato: VUPKHPX Rd Rs1 Rs2

Ejemplo: VUPKHPX m3 m2 m1

Operacion: do i = 0 to 3
$Rd_i <-_{32}$ Extension de signo ($Rs1_i$ [0] ||000|| $Rs1_i$ [1-5] ||000|| $Rs1_i$ [6-10] | 000||$Rs1_i$ [11-15])

VUPKHSB (Vector Byte Unpack High Element)

Cada media palabra del registro destino Rd es la extensión de signo de los bytes de la zona alta del registro fuente multimedia Rs1.

Formato: VUPKHSB Rd Rs1 Rs2

Ejemplo: VUPKHSB m3 m2 m1

Operacion: do i = 0 to 7
$Rd_i <-_{16}$ ExtensionDeSigno ($Rs1_i$)

VUPKHSH (Vector Half Word Unpack High Element)

Cada palabra del registro destino Rd es la extensión de signo de las medias palabras de la zona alta del registro fuente multimedia Rs1.

Formato: VUPKHSH Rd Rs1 Rs2

Ejemplo: VUPKHSH m3 m2 m1

Operacion: do i = 0 to 3
$Rd_i <-_{32}$ ExtensionDeSigno ($Rs1_i$)

VUPKLPX (Vector Pixel Unpack Low Element)

El registro destino multimedia Rd es el resultado de desempaquetar la zona baja del registro fuente multimedia Rs1 extendiendo el signo.

Formato: VUPKLPX Rd Rs1 Rs2

Ejemplo: VUPKLPX m3 m2 m1

Operacion: do i = 0 to 3
$Rd_i <-_{32}$ ExtensionDeSigno ($Rs1_{i+4}$ [0] ||000|| $Rs1_{i+4}$ [1-5] ||000||
$Rs1_{i+4}$ [6-10] | 000||$Rs1_{i+4}$ [11-15])

VUPKLSB (Vector Byte Unpack Low Element)

Cada media palabra del registro destino Rd es la extensión de signo de los bytes de la zona baja del registro fuente multimedia Rs1.

Formato: VUPKLSB Rd Rs1 Rs2

Ejemplo: VUPKLSB m3 m2 m1

Operacion: do i = 0 to 7
$Rd_i <-_{16}$ ExtensionDeSigno ($Rs1_i$)

VUPKLSH (Vector Half Word Unpack High Element)

Cada palabra del registro destino Rd es la extensión de signo de las medias palabras de la zona alta del registro fuente multimedia Rs1.

Formato: VUPKLSH Rd Rs1 Rs2

Ejemplo: VUPKLSH m3 m2 m1

Operacion: do i = 0 to 3
$Rd_i <-_{32}$ ExtensionDeSigno ($Rs1_i$)

VXOR (Vector Logical XOR)

Cada bit del registro destino multimedia Rd es la operación lógica XOR de cada uno de los bits del registro fuente multimedia Rs1 con cada uno de los bits del registro fuente multimedia Rs2.

Formato: VXOR Rd Rs1 Rs2

Ejemplo: VXOR m3 m2 m1

Operacion: $Rd <-_{128}$ Rs1 ? Rs2

Instrucciones Tipo I Multimedia

LVEBX (Byte Vector Load Element Indexed)

Cada operación realiza una carga de un byte en el byte correspondiente de un registro multimedia, el resto de bytes del registro se mantienen indefinidos. La dirección de memoria se calcula sumando el registro fuente entero Rs1 y el registro multimedia Rs2.

Formato: LVEBX Rd,Rs1,Rs2

Ejemplo: LVEBX m0,r4,m7

Operación: DM <- Rs1 + Rs2
I <- Mod(DM,16)
Rd_i <-$_8$ M[DM]

LVEHX (Half Word Vector Load Element Indexed)

Cada operación realiza una carga de media palabra en la media palabra correspondiente de un registro multimedia, el resto de elementos del registro se mantienen indefinidos. La dirección de memoria se calcula sumando el registro fuente entero Rs1 y el registro multimedia Rs2.

Formato: LVEHX Rd,Rs1,Rs2

Ejemplo: LVEHX m0,r4,m7

Operación: s <- 2
DM <- Alineacion(Rs1 + Rs2,s)
I <- Modulo(DM,16) / s
Rd_i <-$_{16}$ M[DM]

LVEWX (Word Vector Load Element Indexed)

Cada operación realiza una carga de una palabra en el elemento correspondiente de un registro multimedia, el resto de elementos del registro se mantienen indefinidos. La dirección de memoria se calcula sumando el registro fuente entero Rs1 y el registro multimedia Rs2.

Formato: LVEWX Rd,Rs1,Rs2

Ejemplo: LVEWX m0,r4,m7

Operación: s <- 4
DM <- Alineacion(Rs1 + Rs2,s)
I <- Mod(DM,16) / s
Rd_i $<-_{32}$ M[DM]

LVSL (VECTOR LOAD FOR SHIFT LEFT)

Cada elemento del registro multimedia destino Rd es el resultado de una permutación tomando para ello los últimos 4 bits de la suma de el registro fuente entero Rs1 y el registro multimedia Rs2.

Formato: LVSL Rd,Rs1,Rs2

Ejemplo: LVSL m0,r0,m2

Operación: DM <- Rs1 + Rs2
Per <- DM[28-31]
if Per = 0x0 then Rd $<-_{128}$ 000102030405060708090A0B0C0D0E0F
if Per = 0x1 then Rd $<-_{128}$ 0102030405060708090A0B0C0D0E0F10
if Per = 0x2 then Rd $<-_{128}$ 02030405060708090A0B0C0D0E0F1011
if Per = 0x3 then Rd $<-_{128}$ 030405060708090A0B0C0D0E0F101112
if Per = 0x4 then Rd $<-_{128}$ 0405060708090A0B0C0D0E0F10111213
if Per = 0x5 then Rd $<-_{128}$ 05060708090A0B0C0D0E0F1011121314
if Per = 0x6 then Rd $<-_{128}$ 060708090A0B0C0D0E0F101112131415
if Per = 0x7 then Rd $<-_{128}$ 0708090A0B0C0D0E0F10111213141516
if Per = 0x8 then Rd $<-_{128}$ 08090A0B0C0D0E0F1011121314151617
if Per = 0x9 then Rd $<-_{128}$ 090A0B0C0D0E0F101112131415161718
if Per = 0xA then Rd $<-_{128}$ 0A0B0C0D0E0F10111213141516171819
if Per = 0xB then Rd $<-_{128}$ 0B0C0D0E0F101112131415161718191A
if Per = 0xC then Rd $<-_{128}$ 0C0D0E0F101112131415161718191A1B
if Per = 0xD then Rd $<-_{128}$ x0D0E0F101112131415161718191A1B1C
if Per = 0xE then Rd $<-_{128}$ 0E0F101112131415161718191A1B1C1D
if Per = 0xF then Rd $<-_{128}$ 0F101112131415161718191A1B1C1D1E

LVSR (Vector Load Shift Right)

Cada elemento del registro multimedia destino Rd es el resultado de una permutación tomando para ello los últimos 4 bits de la suma de el registro fuente entero Rs1 y el registro multimedia Rs2.

Formato: LVSR Rd,Rs1,Rs2

Ejemplo: LVSR m0,r0,m2

Operación: DM <- Rs1 + Rs2
Per $<_4$- DM[28-31]
if Per =0x0 then Rd $<-_{128}$ 101112131415161718191A1B1C1D1E1F
if Per =0x1 then Rd $<-_{128}$ 0F101112131415161718191A1B1C1D1E
if Per =0x2 then Rd $<-_{128}$ 0E0F101112131415161718191A1B1C1D
if Per =0x3 then Rd $<-_{128}$ 0D0E0F101112131415161718191A1B1C
if Per =0x4 then Rd $<-_{128}$ 0C0D0E0F101112131415161718191A1B
if Per =0x5 then Rd $<-_{128}$ 0B0C0D0E0F10111213141516171819 1A
if Per =0x6 then Rd $<-_{128}$ 0A0B0C0D0E0F1011121314151617 1819
if Per =0x7 then Rd $<-_{128}$ 090A0B0C0D0E0F101112131415161718
if Per =0x8 then Rd $<-_{128}$ 08090A0B0C0D0E0F1011121314151617
if Per =0x9 then Rd $<-_{128}$ 0708090A0B0C0D0E0F10111213141516
if Per =0xA then Rd $<-_{128}$ 060708090A0B0C0D0E0F101112131415
if Per =0xB then Rd $<-_{128}$ 05060708090A0B0C0D0E0F1011121314
if Per =0xC then Rd $<-_{128}$ 0405060708090A0B0C0D0E0F10111213
if Per =0xD then Rd $<-_{128}$ 030405060708090A0B0C0D0E0F101112
if Per =0xE then Rd $<-_{128}$ 02030405060708090A0B0C0D0E0F1011
if Per =0xF then Rd $<-_{128}$ 0102030405060708090A0B0C0D0E0F10

LVX (Vector Load Indexed)

Cada operación realiza una carga de 16 bytes de una dirección alineada de memoria en un registro multimedia. Se suma el contenido del registro fuente entero Rs1 con el contenido de el registro multimedia Rs2 y el resultado se alinea tomando el menor valor o igual que sea modulo 16.

Formato: LVX Rd,Rs1,Rs2

Ejemplo: LVX m7,r2,m5

Operacion: DM <- Alineacion(Rs1 + Rs2,16)
Rd $<-_{128}$ M[DM]

LVXL (Vector Load Indexed LRU)

Cada operación realiza una carga de 16 bytes de una dirección alineada de memoria en un registro multimedia. Se suma el contenido del registro fuente entero Rs1 con el contenido de el registro multimedia Rs2 y el resultado se alinea tomando el menor valor o igual que sea modulo 16.

Formato: LVXL Rd,Rs1,Rs2

Ejemplo: LVXL m7,r2,m5

Operacion: DM <- Alineacion(Rs1 + Rs2,16)
Rd $<-_{128}$ M[DM]

VCFSX (Vector Signed Int Convert from Fixed-Point Word)

Cada entero con signo del registro destino multimedia Rd es el resultado de pasar cada entero con signo del registro destino multimedia Rs1 a flotante y dividirlo entre dos elevado a la potencia negativa de cada elemento del registro destino Rs2.

Formato: VCFSX Rd Rs1 Rs2

Ejemplo: VCFSX m3 m2 #14

Operación: do i = 0 to 3
$Rd_i <-_{32} Rs1_i * 2^{-Rs2}$

VCFUX (Vector Unsigned Convert from Fixed-Point Word)

Cada entero sin signo del registro destino multimedia Rd es el resultado de pasar cada entero sin signo del registro destino multimedia Rs1 a flotante y dividirlo entre dos elevado a la potencia negativa de cada elemento del registro destino Rs2.

Formato: VCFUX Rd Rs1 Rs2

Ejemplo: VCFUX m3 m2 #14

Operación: do i = 0 to 3
$Rd_i <-_{32} Rs1_i * 2^{-Rs2}$

VCTSXS (Vector Convert to Signed Fixed-Point Word Saturated)

Cada palabra entera con signo de el registro destino multimedia Rd es el resultado multiplicar cada elemento flotante del registro destino fuente Rs1 por dos elevado a la potencia del registro destino Rs2, saturando el resultado.
Si hay saturación en la multiplicación, el registro VSCR[SAT] se pone a uno.

Formato: VCTSXS Rd Rs1 Rs2

Ejemplo: VCTSXS m0 m1 #5

Operación: do i = 0 to 3
$Rd_i <-_{32}$ Saturacion ($Rs1_i * 2^{Rs2}$)

VCTUXS (Vector Convert to Unsigned Fixed Point Word Saturated)

Cada palabra entera sin signo de el registro destino multimedia Rd es el resultado multiplicar cada elemento flotante del registro destino fuente Rs1 por dos elevado a la potencia del registro destino Rs2, saturando el resultado.
Si hay saturación en la multiplicación, el registro VSCR[SAT] se pone a uno.

Formato: VCTUXS Rd Rs1 Rs2

Ejemplo: VCTUXS m0 m1 #5

Operación: do i = 0 to 3
$Rd_i <-_{32}$ Saturacion ($Rs1_i * 2^{Rs2}$)

VSLDOI (Vector Shift Left Double)

El resultado del registro destino multimedia Rd es la selección de los 16 primeros bytes obtenidos por desplazamiento a la izquierda el valor del registro fuente Rs3 de la concatenación de los registros fuente Rs1 y Rs2.

Formato: VSLDOI Rd Rs1 Rs2 Rs3

Ejemplo: VSLDOI m3 m2 m0 #9

Operacion: do i = 0 to 15
If(I + Rs3 < 16)
then $Rd_i <-_8$ $Rs1_{i+ Rs3}$
else $Rd_i <-_8$ $Rs2_{i+ Rs3}$

VSPLTB (Vector Byte Splat)

Cada byte del registro destino multimedia Rd es el byte del registro fuente multimedia Rs1 especificado en el registro fuente Rs2.

Formato: VSPLTB Rd Rs1 Rs2

Ejemplo: VSPLTB m3 m2 #2

Operacion: j <- Modulo(Rs2,16)
do i = 0 to 15
$Rd_i <-_8 Rs1_j$

VSPLTH (Vector Half Word Splat)

Cada media palabra del registro destino multimedia Rd es la media palabra del registro fuente multimedia Rs1 especificada en el registro fuente Rs2.

Formato: VSPLTH Rd Rs1 Rs2

Ejemplo: VSPLTH m3 m2 #2

Operacion: j <- modulo(Rs2,8)
do i = 0 to 7
$Rd_i <-_{16} Rs1_j$

VSPLTISB (Vector Splat Signed Byte)

Cada byte del registro destino multimedia Rd se obtiene extendiendo el signo al registro fuente Rs1.

Formato: VSPLTISB Rd Rs1

Ejemplo: VSPLTISB m3 #4

Operacion: do i = 0 to 15
$Rd_i <-_8$ ExtensionDeSigno(Rs1)

VSPLTISH (Vector Splat Signed Half-Word)

Cada media palabra del registro destino multimedia Rd se obtiene extendiendo el signo al registro fuente Rs1.

Formato: VSPLTISH Rd Rs1

Ejemplo: VSPLTISH m3 #4

Operacion: do i = 0 to 7
$Rd_i <-_{16}$ ExtensionDeSigno(Rs1)

VSPLTISW (Vector Splat Signed Word)

Cada palabra del registro destino multimedia Rd se obtiene extendiendo el signo al registro fuente Rs1.

Formato: VSPLTISW Rd Rs1

Ejemplo: VSPLTISW m3 #4

Operacion: do i = 0 to 3
$Rd_i <-_{32}$ ExtensionDeSigno(Rs1)

VSPLTW (Vector Word Splat)

Cada palabra del registro destino multimedia Rd es la palabra del registro fuente multimedia Rs1 especificada en el registro fuente Rs2.

Formato: VSPLTW Rd Rs1 Rs2

Ejemplo: VSPLTW m3 m2 #2

Operacion: j <- Modulo(Rs2,4)
do i = 0 to 3
$Rd_i <-_{32} Rs1_j$

STVEBX (Vector Byte Store Element Indexed)

Guarda en memoria un byte de el registro fuente multimedia Rs1 en la dirección de memoria calculada por la suma del registro fuentes entero Rs2 sumado al registro fuente multimedia Rs3. El byte de Rs1 es seleccionado haciendo modulo 16 la dirección calculada.

Formato: STVEBX Rs1 Rs2 Rs3

Ejemplo: STVEBX m1 m2 m3

Operacion: DM <- Rs1 + Rs2
i <- Modulo(DM,16)
$M[DM] <-_8 Rd_i$

STVEHX (Vector Half Word Store Element Indexed)

Guarda en memoria media palabra de el registro fuente multimedia Rs1 en la dirección de memoria calculada por la suma del registro fuentes entero Rs2 sumado al registro fuente multimedia Rs3.La media palabra de Rs1 es seleccionada haciendo modulo 16 la dirección calculada.

Formato: STVEHX Rs1 Rs2 Rs3

Ejemplo: STVEBX m1 m2 m3

Operacion: s <- 2
DM <- Alineacion(Rs1 + Rs2,s)
I <- Modulo(DM,16) / s
M[DM] $<-_{16}$ Rd_i

STVEWX (Vector Word Store Element Indexed)

Guarda en memoria una palabra de el registro fuente multimedia Rs1 en la dirección de memoria calculada por la suma del registro fuentes entero Rs2 sumado al registro fuente multimedia Rs3.La palabra de Rs1 es seleccionada haciendo modulo 16 la dirección calculada.

Formato: STVEWX Rs1 Rs2 Rs3

Ejemplo: STVEWX m1 m2 m3

Operacion: s <- 4
DM <- Alineacion(Rs1 + Rs2,s)
I <- Modulo(DM,16) / s
M[DM] $<-_{32}$ Rd_i

STVX (Vector Store Indexed)

Cada operación guarda en memoria el registro fuente multimedia Rs1 en la dirección calculada por la suma del registro fuente de enteros Rs2 sumado al registro fuente multimedia Rs3. La dirección calculada es alineada al valor menor igual modulo 16.

Formato: STVX Rs1 Rs2 Rs3

Ejemplo: STVX m1 m2 m3

Operacion: DM <- Alineacion(Rs1 + Rs2,16)
M[DM] $<-_{128}$ Rd

STVXL (Vector Store Indexed LRU)

Cada operación guarda en memoria el registro fuente multimedia Rs1 en la dirección calculada por la suma del registro fuente de enteros Rs2 sumado al registro fuente multimedia Rs3. La dirección calculada es alineada al valor menor igual modulo 16.

Formato: STVXL Rs1 Rs2 Rs3

Ejemplo: STVXL m1 m2 m3

Operacion: DM <- Alineacion(Rs1 + Rs2,16)
M[DM] $<-_{128}$ Rd

Funciones

Alineacion(x,y):
Alinea el valor de la dirección x a y.

Modulo(x,y):
Resto de x entre y.

CarryOut(x+y):
Carry out de la suma x + y

Maximo(x,y):
Selecciona el valor máximo entre x e y.

Minimo(x,y):
Selecciona el valor mínimo entre x e y.

Reciproco(x):
Calcula el reciproco del valor de x.

Suelo(x):
Redondea al valor entero menor o igual que x.

Cercano(x):
Redondea al valor entero más cercano que x.

Ceil(x):
Redondea al valor entero mayor o igual que x.

Truncamiento(x):
Trunca x quedándose solo con la parte entera.

Rotar(x,y):
Rota hacia la izquierda los bits de x y veces.

ReciprocoRaiz(x):
Reciproco de la raíz cuadrada de x.

DesplazamientoIzquierda(x,y):
Desplaza x hacia la izquierda tantas veces como indique y.

DesplazamientoDerecha (x,y):
Desplaza x hacia la derecha tantas veces como indique y.

ExtensionDeSigno(x):
Extiende el signo de x.